

Advanced Tools for Mission Editor

ATME - Module Development Reference Manual

Author Sunski34

Traduction and usage tests Snowsniper

Contents

Introduction.....	14
Callback concept.....	15
ATME events concept	16
ATME & DCS mission editor	17
Definitions and notations.....	18
Restrictions & limitations.....	19
ATME user modules creation	20
Module 1 : User module : « Hello world ».....	20
Lua code	20
Explanations	21
Tests	21
Exercice	25
Module 1 : Adding a dynamic display	25
Lua code	25
Explanations	26
Tests	26
Exercice	26
Module 1 : dynamic display replacement with a specific user Menu.....	27
Lua code	27
Explanation.....	28
Tests	28
Exercice	29
Module 1 : Adding a menu visible and available in some cases.....	30
Lua code	30
Explanations	31
Tests	32
Exercice	33
Module 1 : Add a submenu and a player specific variable	34
Lua code	34

Explanations	35
Tests	36
Exercice	36
Module 2 : DCS Interactions and User Trigger Management.....	37
Lua code	37
Explanations	38
Tests	39
Exercice	40
Module 2 : User Triggers management with a Flag.....	40
Lua code	41
Explanations	42
Tests	42
Exercice	42
Module 2 : User triggers with time Management.....	43
Lua code	43
Explanations	44
Tests	45
Exercice	45
Module 3 : Mission data for groups and dynamic group creation	46
Lua code	46
Explanations	47
Tests	47
Exercice	48
Module 4 : Core Events management	49
Lua code	50
Explanations	51
Tests	51
Exercice	52
Module 4 : pickupzones Management	54
Lua code	54
Explications	55
Tests	55
Exercice	56
Module 4 : Multilangage management	57

Lua code	58
Explanations	59
Tests	60
Module 5 : Race management.....	62
Lua Code.....	62
Explanations	64
Tests	64
Global functions of the ATME table.....	66
ATME.arePointsVisible(pA, pB)	66
ATME.convertFtstoM(distance)	66
ATME.convertKphtoMps(speed).....	67
ATME.convertLLToMGRS(latitude, longitude)	67
ATME.convertLLToPoint(latitude, longitude, altitude)	68
ATME.convertMGRSToLL(tableMGRS).....	68
ATME.convertMpstoKph(speed).....	69
ATME.convertMpstoNds(speed).....	69
ATME.convertMtoFts(distance)	69
ATME.convertMtoNM(distance).....	70
ATME.convertNdstoMps(speed).....	70
ATME.convertNMtoM(distance).....	70
ATME.convertPointToLL(point).....	71
ATME.convertToDMS(angle, secDec)	71
ATME.convertToHMSd(seconds)	72
ATME.convertToPoint3D(point).....	72
ATME.displayForAll(text, duration).....	72
ATME.displayForCoalition(coalitionName, text, duration)	72
ATME.displayForCountry(countryName, text, duration)	73
ATME.explode(point, power).....	74
ATME.getAIUnits().....	74
ATME.getFlag(id).....	74
ATME.getGroups().....	75
ATME.getLanguage()	75
ATME.getPlayers().....	75
ATME.getPointAltitude(point)	76

ATME.getSurfaceAltitude(point).....	76
ATME.getTheatre().....	76
ATME.getTime()	77
ATME.getWindAzimuth(point3D, withTurbulence).....	77
ATME.getWindHSpeed(point3D, withTurbulence).....	77
ATME.getWindSpeed(point3D, withTurbulence)	78
ATME.getWindVector(point3D, withTurbulence).....	78
ATME.getDCSZone(zoneName).....	79
ATME.isObjectClass(object, class).....	79
ATME.isPoint(tableToVerify).....	80
ATME.isStaticObject(objectName).....	80
ATME.isSurfaceLand(point).....	80
ATME.isSurfaceRoad(point)	81
ATME.isSurfaceRunway(point).....	81
ATME.isSurfaceWater(point)	81
ATME.isUnitControllable(unitName)	82
ATME.loopTransmission(file, point, modulation, frequency, power)	82
ATME.rotationH(point, center, angle)	83
ATME.run(language)	84
ATME.scaireHFromPoints(pA, pB, pC).....	84
ATME.sendTransmissionOnce(file, point, modulation, frequency, power).....	85
ATME.setDebugLevel(level)	85
ATME.setF10groupIDAlphaOrder()	85
ATME.setF10AlphaOrder()	86
ATME.setFlag(id, value).....	86
ATME.soundForAll(file)	86
ATME.soundForCoalition(coalition, file)	87
ATME.soundForCountry(countryName, file)	87
ATME.startTransmission(file, point, modulation, frequency, power, interval).....	88
ATME.stopTransmission(id)	88
ATME.whichSide(pA, pB, pC)	89
Fonctions de création d'objets statiques de la table ATME	90
ATME.staticObjects.createCargo(countryName, name, type, position, mass, isDead).....	90
ATME.staticObjects.createWhiteContainer(countryName, name, position, heading, isDead)..	91

ATME Class	92
ATME.C_AIUnit class.....	92
ATME.C_AIUnit.exists(name)	92
ATME.C_AIUnit.getByName(name)	92
Object:crossAxisFromLeftToRight(pA, pB)	93
Object:crossAxisFromRightToLeft(pA, pB)	93
Object:disable()	94
Object:explode(power)	94
Object:fireFlare(color).....	94
Object:fireIlluminationBomb(power).....	95
Object:fireSmoke(color).....	95
Object:getAGLAltitude().....	95
Object:getAzimuth().....	96
Object:getCallsign()	96
Object:getClassName().....	96
Object:getCoalitionName()	97
Object:getCountryName().....	97
Object:getDCSUnit()	98
Object:getFuelRatio()	98
Object:getGroup()	98
Object:getGroupsNameOnBoard()	99
Object:getHSpeed().....	99
Object:getLife()	99
Object:getLifeRatio()	100
Object:getMSLAltitude()	100
Object:getName().....	100
Object:getNbUnitsOnBoard().....	100
Object:getNearestReadyToBoardGroups(radius)	100
Object:getPosition()	101
Object:getRollAxisVector().....	101
Object:getSpeed()	102
Object:getSpeedAzimuth().....	102
Object:getTypeName().....	102
Object:getVelocityVector()	102

Object:getVSpeed()	102
Object:inAir()	103
Object:isAAA()	103
Object:isAirDefence()	104
Object:isGroundVehicle()	104
Object:isHelicopter()	104
Object:isInfantry()	105
Object:isInDCSZone(zoneName)	105
Object:isInZone2D(reference, radius)	106
Object:isInZone3D(reference, radius)	107
Object:isNear(reference, radius, deltaAltitude)	108
Object:isManPad()	109
Object:isPersonnelCarrier()	109
Object:isPlane()	109
Object:isSAMVehicle()	110
Object:isSAMSiteCommandCenter()	110
Object:isSAMSiteLauncher()	110
Object:isSAMSiteRadar()	111
Object:load(groupToBoard, radius)	112
Object:loopTransmission(file, modulation, frequency, power)	113
Object:sendTransmissionOnce(file, modulation, frequency, power)	113
Object:startTransmission(file, modulation, frequency, power, interval)	114
Object:stopAllTransmissions()	114
Object:stopTransmission(id)	115
Object:unload(id)	115
Object:whichSide(reference)	116
Classe ATME.C_AirBase	117
Object:getClassName()	117
Object:getName()	117
ATME.C_EventMgr Class	118
Object:getCoreEventType(id)	118
Object:getCoreEventDatas(id)	120
Object:isCoreEvent(id)	124
Object:isUserTriggerEvent(id)	125

Object:isUserTriggerEventToggle(id)	125
Object:pairs().....	126
ATME.C_Flare class	126
ATME.C_Flare(colorName, point)	126
Object:fire(duration)	126
Object:fireOnce().....	126
Object:getClassName().....	127
Object:getColor().....	127
Object:getDuration()	127
Object:getPoint().....	128
Object:getTimeStart()	128
Object:stop()	128
ATME. C_Group Class.....	129
ATME.C_Group.exists(name)	129
ATME.C_Group.getByname(name)	129
ATME.C_Group.getGroupsInDCSZoneForAll(zoneName, allGroup)	130
ATME.C_Group.getGroupsInDCSZoneForCoalition(coalitionName, zoneName, allGroup)	130
ATME.C_Group.getGroupsInZone2DForAll(reference, radius, allGroup)	131
ATME.C_Group.getGroupsInZone2DForCoalition(coalitionName, reference, allGroup)	132
ATME.C_Group.getReadyToBoardGroupsForAll()	133
ATME.C_Group.getReadyToBoardGroupsForCoalition(coalitionName, zoneName, allGroup)	133
Object:changeReadyToBoard(value)	133
Object:activate().....	133
Object:disable()	134
Object:freeze(on)	134
Object:getBarycentre()	135
Object:getCategoryName()	135
Object:getClassName().....	135
Object:getCoalitionName()	135
Object:getDCSGroup().....	135
Object:getFirstUnit()	136
Object:getLastUnit()	136
Object:getMaxDistance(reference).....	136
Object:getMaxHDistance(reference)	137

Object:getMinDistance(reference)	137
Object:getMinHDistance(reference).....	138
Object:getName().....	138
Object:getNbUnits()	139
Object:getNextUnit().....	139
Object:getPickupZone().....	139
Object:getPreviousUnit()	140
Object:getUnitByName(name).....	140
Object:getUnits().....	140
Object:hasPickupZone()	141
Object:isActivated().....	141
Object:isBoardingStarted().....	142
Object:isInDCSZone(zoneName, allGroup)	142
Object:isInZone2D(reference, radius, allGroup)	143
Object:isInZone3D(reference, radius, allGroup)	144
Object:isNear(reference, radius, deltaAltitude, allGroup).....	145
Object:isOnlyInfantry().....	146
Object:isOnlyAI()	146
Object:isReadyToBoard()	146
Object:isSignalSet()	147
Object:isStopped()	147
Object:move()	147
Object:resetPickupZone()	148
Object:resetSignal().....	148
Object:setPickupZone(zoneName, random).....	149
Object:setRoute(...)	150
Object:setSignal(radius, signalType)	151
Object:stop()	152
ATME. C_GroupSpawnDatas class	152
ATME.C_GroupSpawnDatas.duplicateFromMissionDatas(groupName, newGroupName)	152
Object:getClassName().....	152
Object:getName().....	153
Object:spawn(...)	153
ATME.C_IndexList class	155

ATME.C_IndexList()	155
Object:add(item)	155
Object:get(index)	155
Object:getClassName()	156
Object:getCount()	156
Object:remove(index)	156
Object:removeAll()	156
Object:pairs()	156
ATME.C_Line2D class	158
ATME.C_Line2D(...)	158
Object:get()	160
Object:getA()	160
Object:getB()	160
Object:getC()	160
Object:getClassName()	161
Object:getOrigine()	161
Object:getPerpendicular(offset)	162
Object:getPointFromOrigine(offset)	162
Object:getX()	163
Object:getZ()	163
Object:isNSAxis()	165
Object:isWEAxis()	165
ATME.C_MenuF10 Class	166
ATME.C_MenuF10(player, label, parent)	166
Object:append(groupId, menuLabel, radioHandler, argsRadioHandler)	166
Object:remove(groupId)	166
Object:removeAll()	167
ATME.C_Module Class	168
ATME.C_Module(name, handlers, debugOn)	168
Object:error(text)	171
Object:getClassName()	171
Object:isDebugOn()	171
Object:output(text, level)	172
Object:createAbsoluteUserTrigger(name, condition)	172

Object:createFlagRelativeUserTrigger(name, flagNumber, condition)	173
Object:getUserTriggerByName(name).....	175
Object:removeUserTrigger(name)	175
Object:userTriggerExists(name).....	175
ATME. C_Player class	176
ATME.C_Player.exists(name)	176
ATME.C_Player.getByname(name)	176
Object:crossAxisFromLeftToRight(pA, pB).....	177
Object:crossAxisFromRightToLeft(pA, pB).....	177
Object:display(text, duration)	178
Object:explode(power)	178
Object:getAGLAltitude().....	178
Object:getAzimuth().....	179
Object:getCallsign().....	179
Object:getClassName().....	179
Object:getCoalitionName()	180
Object:getCountryName().....	180
Object:getDCSUnit()	181
Object:getF10MenuRoot()	181
Object:getFuelRatio()	181
Object:getGroup()	182
Object:getGroupsNameOnBoard()	182
Object:getHSpeed().....	182
Object:getLife()	183
Object:getLifeRatio()	183
Object:getMSLAltitude()	183
Object:getName().....	184
Object:getNbUnitsOnBoard().....	184
Object:getNearestReadyToBoardGroups(radius)	184
Object:getPosition()	185
Object:getPseudo()	185
Object:getRollAxisVector().....	185
Object:getSpeed()	186
Object:getSpeedAzimuth().....	186

Object:getTypeName()	186
Object:getVelocityVector()	187
Object:getVSpeed()	187
Object:inAir()	187
Object:isEngineStarted()	188
Object:isGroundVehicle()	188
Object:isHelicopter()	188
Object:isInDCSZone(zoneName)	189
Object:isRouteInDirection(reference)	189
Object:isInZone2D(reference, radius)	190
Object:isInZone3D(reference, radius)	191
Object:isNear(reference, radius, deltaAltitude)	192
Object:isPersonnelCarrier()	193
Object:isPlane()	193
Object:load(groupToBoard, radius)	193
Object:soundOnce(file)	195
Object:soundChange(file, interval, forced)	195
Object:soundPause(duration)	195
Object:soundStart(file, interval)	196
Object:soundStop()	196
Object:soundChangePlayList(playList, interval, forced)	196
Object:soundStartPlayList(playList, randomRead, interval)	196
Object: unload(id)	197
Object:whichSide(reference)	198
Classe ATME.C_Race	200
ATME.C_Race(name, leftSideName, rightSideName, nbTurns)	200
ATME.C_Race.exists(name)	201
ATME.C_Race.getByName(name)	201
Object:addPlayer(player)	201
Object:delete()	202
Object:displayForAllPlayers(text, duration)	202
Object:isPlayerInRace(player)	202
Object:getClassName()	203
Object:getDoorIndex(offset)	203

Object:getDoorSides(index)	203
Object:getLapAtDoor(index)	204
Object:getMaxAltitude()	204
Object:getMaxAltitudePenalty()	204
Object:getMissedDoorPenalty().....	205
Object:getName().....	205
Object:getNbDoors()	205
Object:getNbTurns()	205
Object:getPlayerNextDoorIndex(player).....	206
Object:getRanking()	206
Object:removeAllPlayers()	207
Object:removePlayer(player).....	207
Object:resetMaxAltitudeRule()	207
Object:resetMissedDoorRule()	207
Object:setLapAtDoor(index)	208
Object:setMaxAltitudeRule(maxAltitude, penalty).....	208
Object:setMissedDoorRule(penalty).....	208
Object:soundForAllPlayers(file)	209
ATME.C_Smoke class	210
ATME.C_Smoke(colorName, point)	210
Object:activate(restart).....	210
Object:getClassName().....	211
Object:getColor().....	211
Object:getPoint().....	211
Object:stop()	211
ATME.C_StaticObject class	212
ATME.C_StaticObject.exists(name).....	212
ATME.C_StaticObject.getByName(name).....	212
Object:explode(power)	212
Object:fireFlare(color).....	212
Object:fireIlluminationBomb(power).....	213
Object:fireSmoke(color).....	213
Object:getAzimuth().....	214
Object:getClassName().....	214

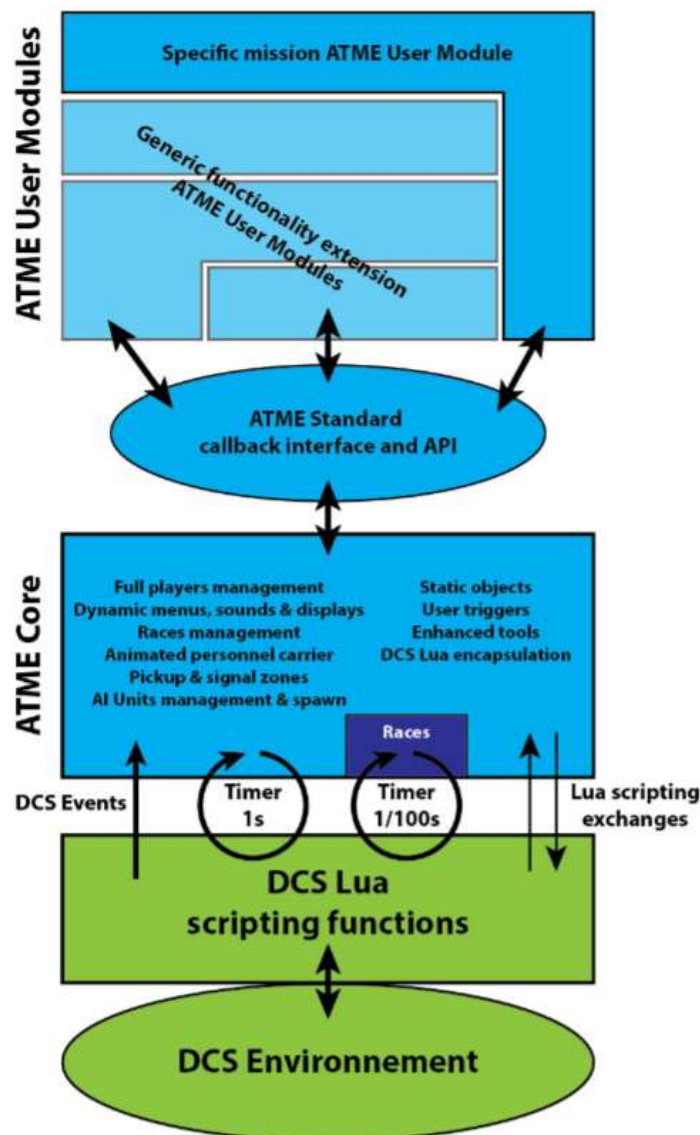
Object:getCoalitionName()	214
Object:getCountryName().....	215
Object:getDCSStaticObject()	215
Object:getLife()	216
Object:getName().....	216
Object:getPosition()	216
Object:getTypeName().....	217
ATME.C_UserTrigger class.....	217
Object:getClassName().....	217
Object:getFlag().....	218
Object:getName().....	218
Object:getTimeEnd()	218
Object:getTimeStart()	219
Object:isActivated().....	219
Object:isArmed().....	219
ATME.C_Vector3D class	220
ATME.C_Vector3D(...)	220
Object:get()	221
Object:getAzimuth().....	221
Object:getClassName().....	221
Object:getHModule()	222
Object:getModule().....	222
Object:getX()	222
Object:getY()	223
Object:getZ()	223
Object:toUnitVector()	223
Object:scalaireH(vector)	224

Introduction

Advanced Tools for Mission Editor or ATME is a set of lua classes and functions which intend to simplify DCS World mission editor Lua script implementation.

ATME aim is to simplify the creation of single and multiplayer compatible missions, without limiting the number of players. Players entering, destruction or exit mission, the ability of changing slots are fully implemented.

ATME is based on a module concept, the basic module is called ATME Core and have to be loaded imperatively first. Each additional module or ATME User Module created can be generic or dedicated to a mission. Future generic modules will provide additional functionalities. This concept offers great flexibility, allowing users who are not comfortable with Lua to create complex interactiv missions with few code.



Special focus has been used to avoid unintentional data corruption with the interdiction to modify certain object data. Special functions are available for this purpose and should be used. The inside objects data are not directly accessible to the creators of user modules.

The **Core ATME** module brings all the necessary abstraction and interfaces with many of DCS World's functions. These functions organization is primarily oriented towards mission designers. An object approach provides the necessary clarity.

ATME Core brings together the following major features:

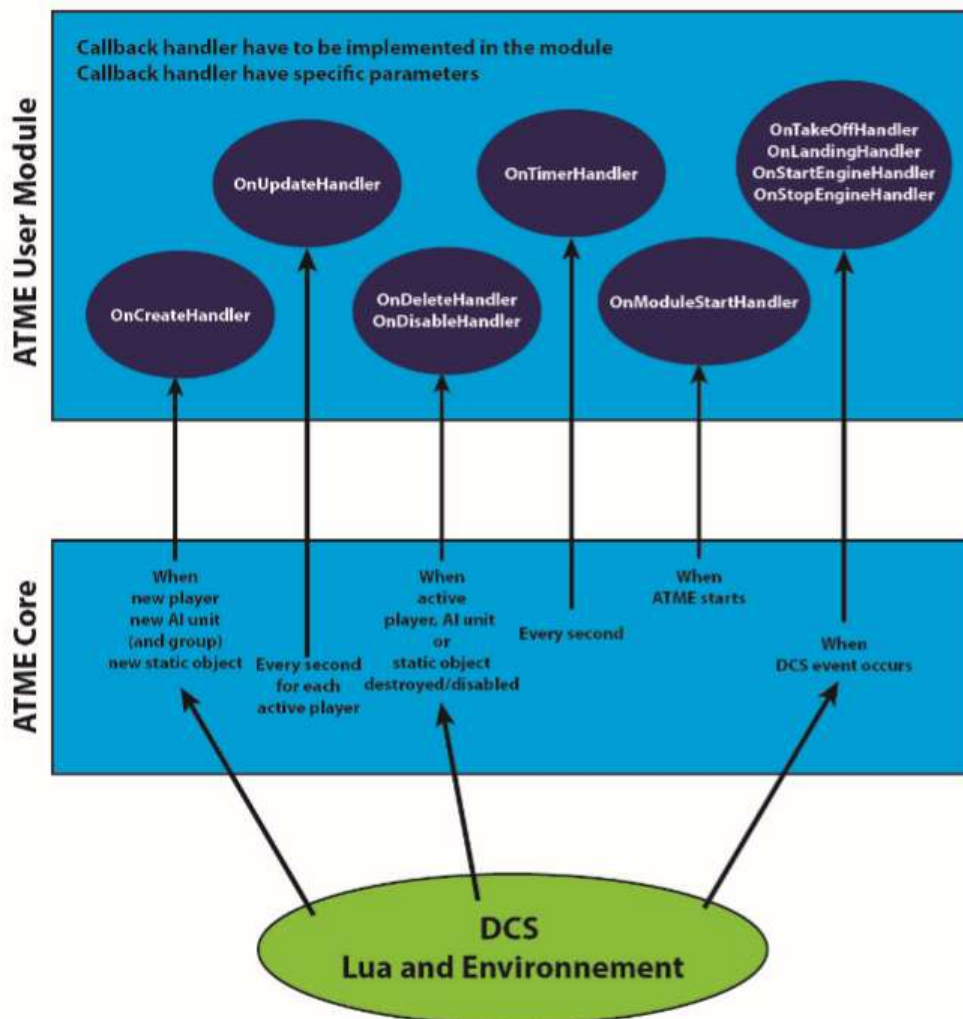
- Providing global functions including DCS functions (encapsulation), generic (conversions, etc ...) or mathematics (3D vectors, 2D lines, relative positioning, etc.)
- Referencing mission based FARP & Airfields for a theater operation (NEVADA or CAUCASUS for the moment).
- Player objects, AI units, groups and static objects management. Spawn ability, roads or recoveries of route from a givwaypoint association etc ...
- Adding specific objects to facilitate mission and interactiv management (radio players menu, smoke, flares, etc ...)
- Multiplayer races with 1/100th, intermediate, lap, total time and rankings, complete management. Creating raids with several races ability.
- Infantry units with dynamic climb in the vehicle, embarkation / disembarkation complete management. Transport of troops (personnel carrier) concept, authorizing the AI unit embarkation on board (ground or helicopters) or player unit (helicopters). Able to creating embarkation zones with smoke or flare position reports.
- Core events triggering which can be processed in the modules (end of embarkation, race events, reporting zone entry into, etc.).
- User triggers creation and management allowing triggering of a single or repeated event at a "T" time which can be defined in absolute time, or relative to a flag. This User trigger Event will only be available to the module that created the trigger.
- Multi-language support: French, English, German and Russian. Available currently for message and label. Multi-language implementation is the module creator choice & responsibility.

ATME is DCS 1.5.5 (and later) & DCS 2.0.4(and later) compatible.

Callback concept

This concept allows the ATME core module to communicate with ATME user modules, bringing the necessary dynamics. A callback is a function defined and implemented in the user modules. It will be called by the ATME Core module depending on the mission context. These functions have defined and imposed parameters. It concerns objects creation, destruction or deactivation, processing particular DCS events such as take-off or landing, or the every second regular activation internal

ATME Core timer (for callbacks onUpdate Or onTimer). These callbacks must be known to the Core ATME module, so any new module will have to declare its callbacks via a dedicated handlers list. A handler is in fact a function identifier allowing the ATME Core module to call function when needed.



Callbacks may exist for each family : Players, AI Units, Groups or static Objects.

Example :

OnCreatePlayerHandler,
OnCreateAIUnitHandler,
OnCreateGroupHandler,
OnCreateStaticObjectHandler

ATME events concept

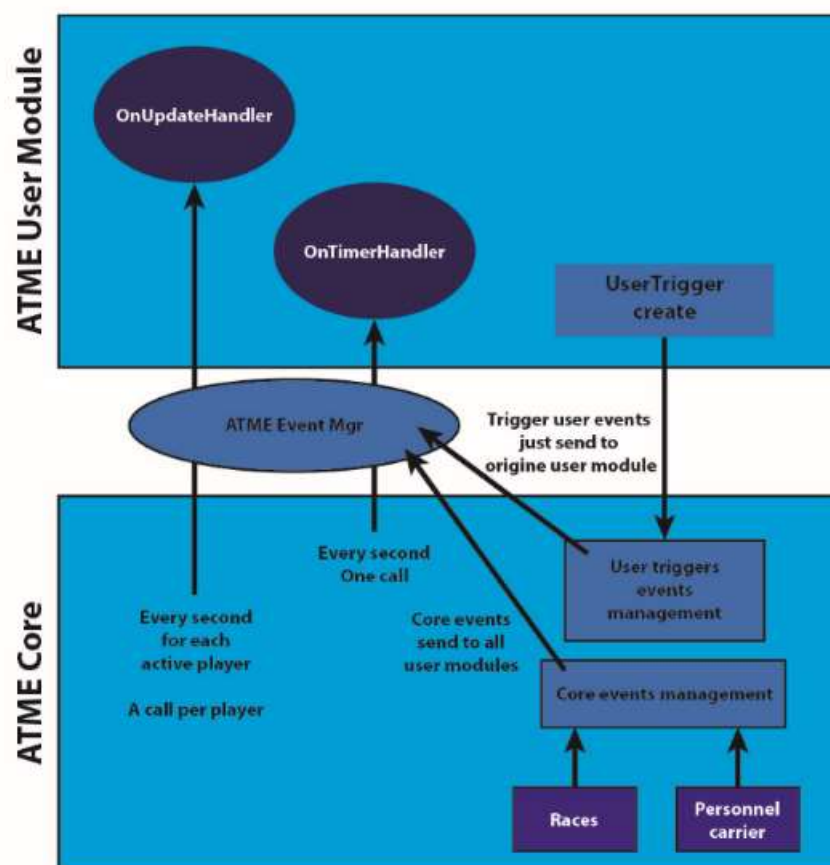
ATME events are transmitted through two callbacks related to the ATME internal timer. This timer is triggered every second.

There are two ATME events types:

- ATME Core events: These are the result of processing in the Core ATME module. These include events related to troop transport, reporting areas management or race management. These Core events are sent to all current mission user modules.

- Events triggered by User Trigger Event: A user trigger triggers an event at a givtime of the mission for a specified duration. This duration may be zero, in which case the event will only be issued once. A User Trigger must be created beforehand in a user module. Events generated by User Triggers will be sent only to the user modules that created them.

These events are built into an ATME object named ATME Event Mgr (Manager). This instance will be passed to two callbacks : OnUpdate and OnTimer. This instance will only contain events that are active at the time of the call.



ATME Event manager (Mgr) is OnUpdateHandlers and OnTimerHandlers parameter.

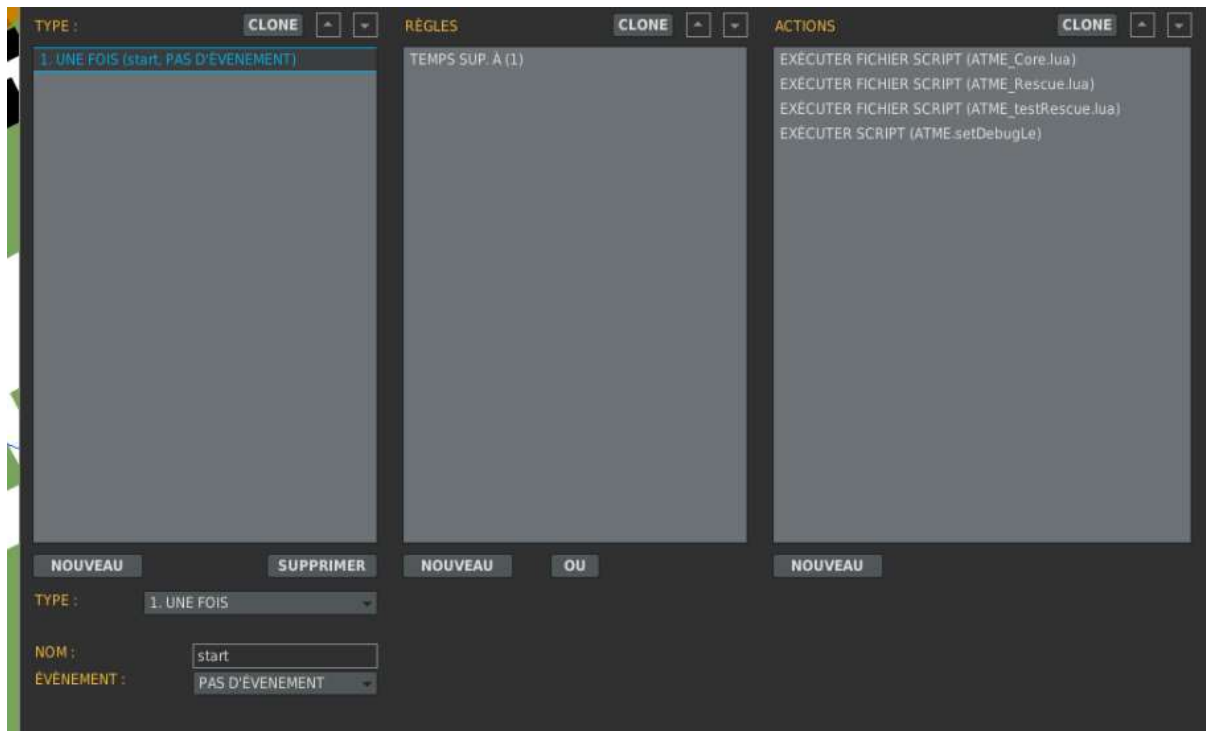
Each event is characterized by its source (Core or Trigger User), its Core events type, and its associated variable data depending on the event. These data are gathered in a table and made available to the creator of the module by appropriate functions.

ATME & DCS mission editor

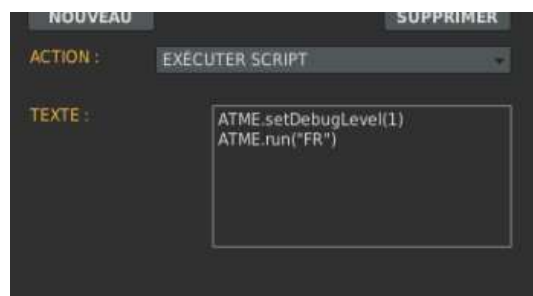
ATME modules are integrated into a mission, like all lua script files.

In Mission Editor :

- Create a trigger that triggers after 1 second and load all the necessary to operate ATME environment. Scripts files of the ATME modules : Core first followed by the user modules themselves in the appropriate order.



- A script created in the editor that triggers ATME by its **ATME.run function**. It also specifies the desired language (if supported by declared user modules). The `ATME.setDebugLevel` function is only useful for testing.



It is also possible to create global functions or variables in a user module (ATME User module) that can be called directly through mission editor. ATME Core makes available to the module creator a global **table** per module, referenced under the name **ATME.modules["module name"]**

Definitions and notations

A ATME class is defined by :

- **Class** : Structure defining the objects properties managed by ATME and / or derived from DCS. We will find a particular class, for players, AI units, groups but also more abstract classes such as modules or 3D vectors.
- **Instance** : An instance is the concrete representation of a particular object in the mission or in ATME. An instance is necessarily attached to one and only one class in ATME. Thus, during a mission, there is a multitude of instances of all classes. An instance can be created or destroyed. Some instances associated with particular classes are completely managed by ATME (players, AI units, groups, static object); It is not possible to create them for the creator of a module.

Following terminology should be used throughout this manual :

- Standard ATME functions named **ATME.function_name** ou **ATME.staticObject.spawn_function_name**
- The instance construction function for a givclass, if possible, is **ATME.Class_Name**. This particular function returns a lua table corresponding to a new instance of the class. **This function is called a constructor.**
- **ATME. Class_Name.class_function** functions: These functions do not refer to a particular class instance. For base objects referencing players, AI units, groups, static objects, or runs, **getByName** (or **exists**) is used to retrieve the object instance (or to verify its existence). Other functions however, exist.
- Object instance methods (or functions) **Object:instance_function** related to a particular object instance. **Beware of this syntax.**
- Associated attributes (variables) are accessible only with particular functions. Certain properties are not allowed to the mission creator because used only for the internal operation of ATME.

These three notations will be used in this manual:

- **ATME.Class_Name**
- **ATME. Class_Name.class_function**
- **Object:instance_function**

Restrictions & limitations

ATME has not betested or validated with the DCS Combined Arms. Also, there is no guarantee that a player will be able to control the vehicle on the ground. These tests will be carried out later.

ATME user modules creation

The creation of an ATME module will be presented with increasing complexity examples. For more information, refer to the global functions & ATME classes definition explained later in this manual. Indeed, in these examples, the in-depth description of the functions used will not be explained.

ATME Core encapsulates all the player management, AI units, racing and other embarkation / disembarkation possibilities, dynamic player menus, reporting area etc. Also the implementation of a new module still easy :

- Creating a lua script file
- Module framework implementation of the callbacks and creation of the module instance class allowing the support of the module by ATME Core.
- Variables and functions implementation inside the module
- Global functions and variables declaration and implementation which can be used directly in the DCS mission editor or in another module.

Module 1 : User module : « Hello world »

This very simple module allows the display of "Hello World ..." followed by the pseudo multiplayer ("new nickname" in single player) as soon as a player takes control of an airplane or a helicopter. For clarity, unused callback handlers are set to **nil**. This is not required, since any uninitialized lua variable is **nil**. But this allows to visualize the exhaustive list of the existing callback handlers in ATME.

This module implements only one callback linked to the mission player entry. it will use this ATME module.

Lua code

```
-- Advanced Tools for Mission Editor
local thisModule

local function onCreatePlayer(player)
    player:display("Hello World ... " .. player:getPseudo(), 10)
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
    }
```

```

onDisableAIUnitHandler = nil,
onTakeoffAIUnitHandler = nil,
onLandingAIUnitHandler = nil,
onStartEngineAIUnitHandler = nil,
onStopEngineAIUnitHandler = nil,

onCreateGroupHandler = nil,
onDeleteGroupHandler = nil,
onDisableGroupHandler = nil,

onCreateStaticObjectHandler = nil,
onDeleteStaticObjectHandler = nil,

onTimerHandler = nil,
onModuleStartHandler = nil,
}

thisModule = ATME.C_Module("HelloWorld", newHandlers, true)
end

```

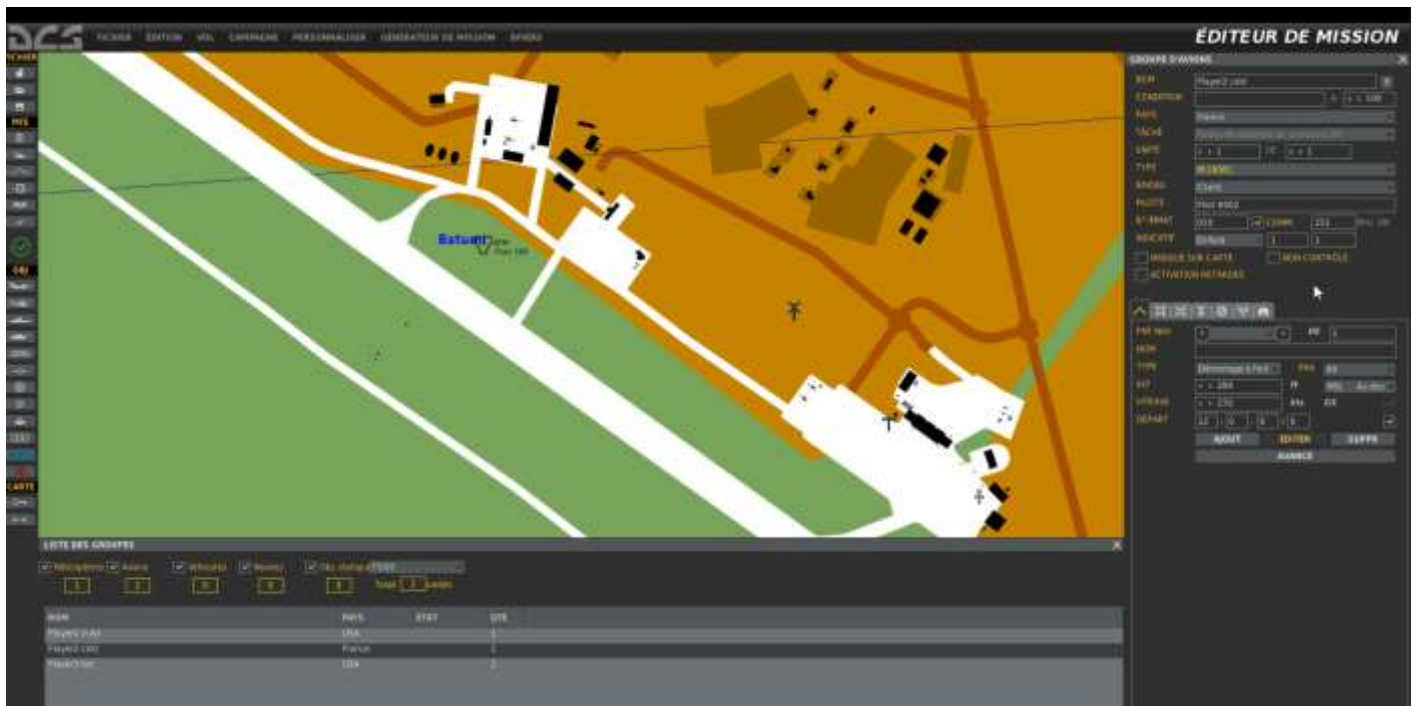
Explanations

This module consists of three parts:

- global variable definition inside the module: **thisModule**. This variable is the **ATME.C_Module** instance for this module. It will be initialized in the 3rd part. It will be used by functions of the **ATME.C_Module** class.
- Callback definition **onCreatePlayer** : parameter callback type is imposed by ATME Core ; **player** is a **ATME.C_Player** instance representing player units. As soon as a player arrives in the mission, this callback will be called. **display** & **getPseudo** are **ATME.C_Player** class functions. Reference is made to the definition of ATME classes later in this manual. **player:display** is a lua object notation. This indicates that we want to use the **display** function for the **player** instance. This notation will always be used. For more information, reference should be made to the literature on the existing lua language on the Internet. **The use of `player.display` instead of `player:display` will cause an error.**
- Module Initialization after the comment **--MAIN** is encarted by **do .. end** for clarity : This part is executed as soon as the lua script file is declared in the mission editor (EXECUTE FILE SCRIPT). It defines the callback handlers table (here only the handler of the **onCreatePlayer** callback defined in Part 2. This handler will be copied to the newHandlers table (**onCreatePlayerHandler**). Other callback handlers are not used in this module, their value will be **nil**. Finally, the module instance is initialized by the constructor **ATME.C_Module**. The first parameter is the module name that must be unique for a givmission, the second parameter is the handlers table which will be used by ATME. The last parameter, (here true), indicates whether a module is in debug mode and should be set to false once the module has befinalized and tested. The module is now finished. It is possible to enrich it or change the displayed message using other functions existing in ATME.

Tests

For testing, create a mission and define units as a client. Units number does not matter, because everything will be supported by ATME.

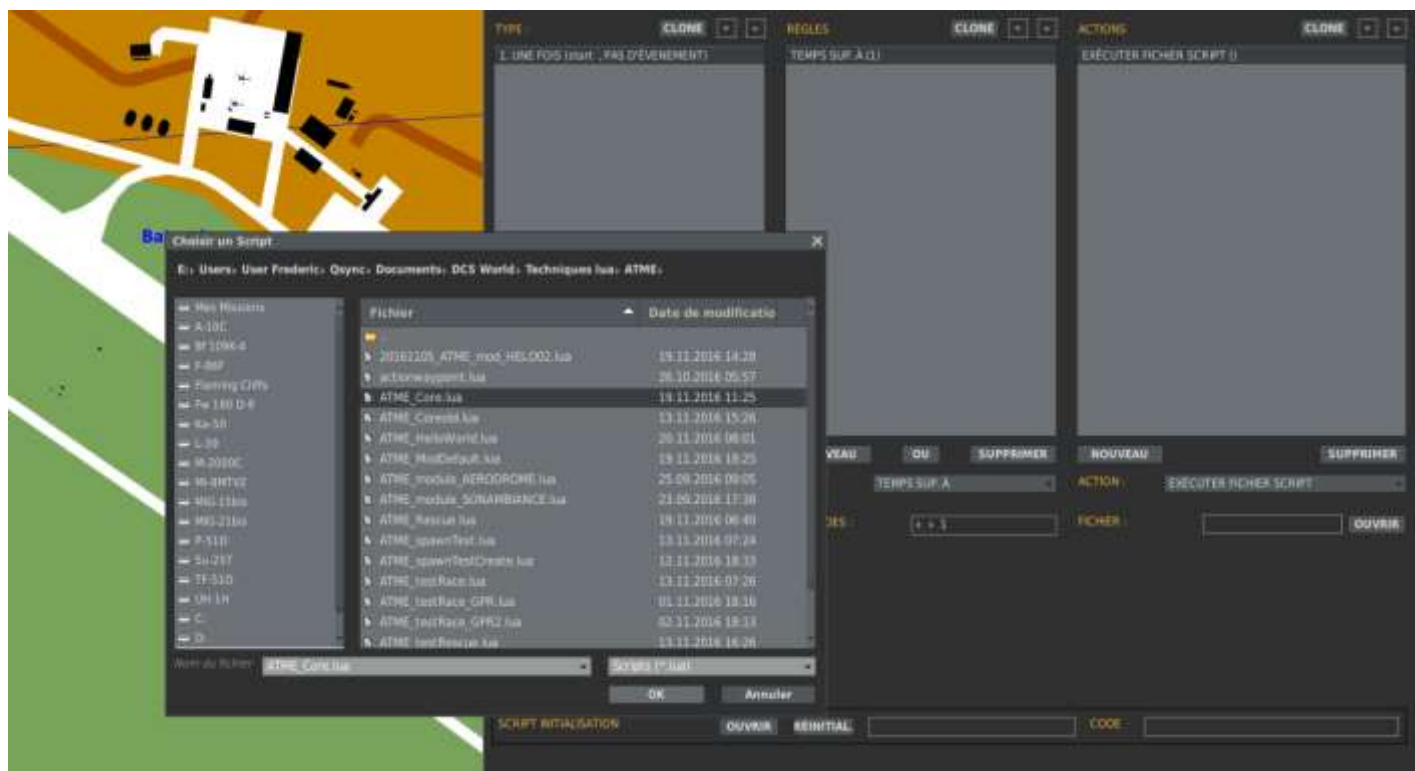


In the example above, 3 player units were created, two ground planes and one helicopter in flight. It is of course possible to create player units for blue or red coalitions. For this module, no distinction is made. All players will see the message created.

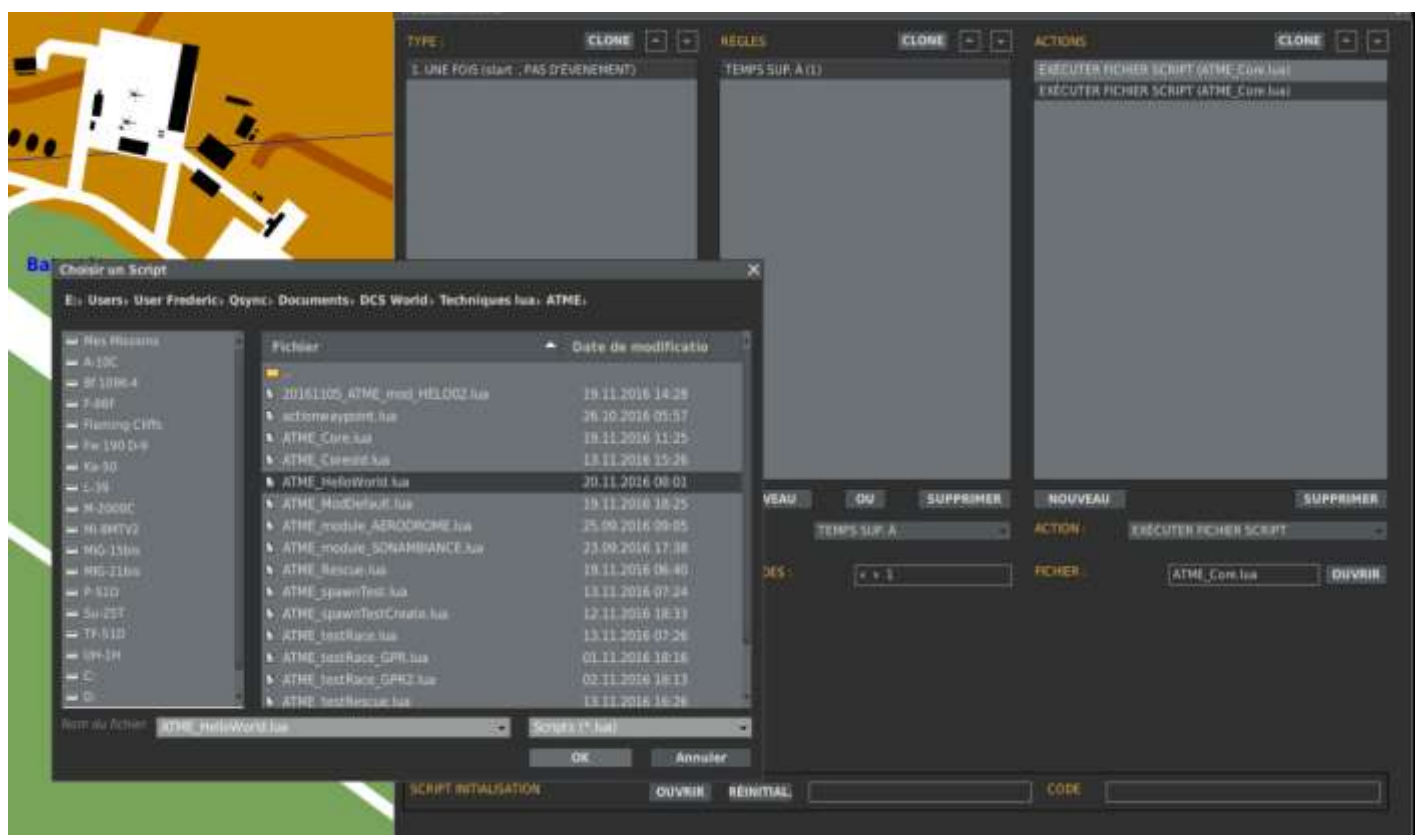
You have to add the trigger to start ATME, here named start:



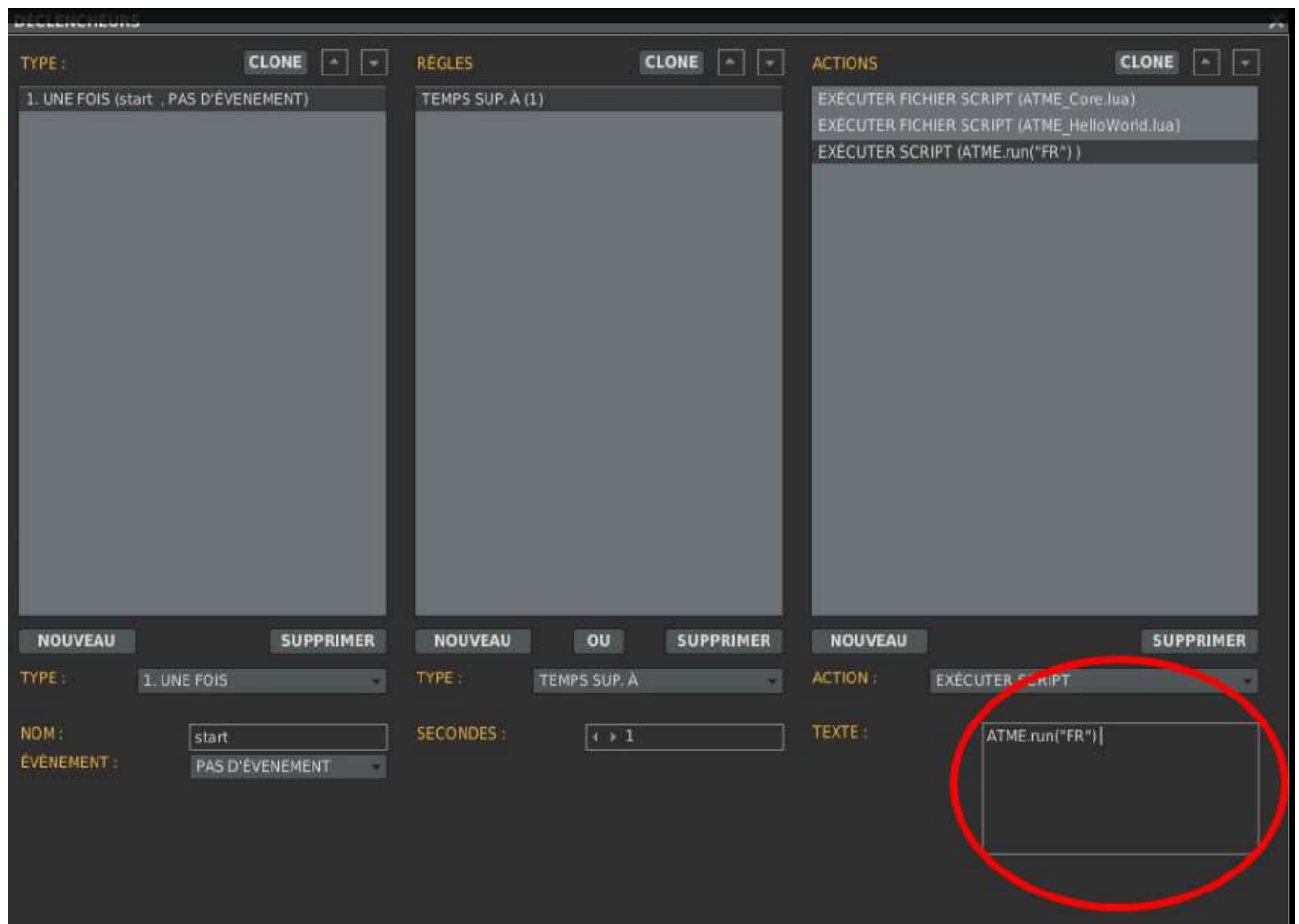
Thset a rule for a trigger after 1s, and finally define the "EXECUTE FILE SCRIPT" action for ATME Core:



The following must be cloned for the HelloWorld module:



Finally, create the script containing the function **ATME.run** (and if needed the **ATME.setDebugLevel** function, not done here):



Therefore, ATME is ready to be used in this mission with the module created, below a test in single player:



Result will be identical in multiplayer displaying the player nickname. It is also possible to do the same in version 2.0.4 and following, on the NEVADA theater.

Exercise

Exercise : Display « Welcome in ... » and the Theatre name in place of « Hello World ». **ATME.getTheatre()** function will be used.

Soluce :

```
local function onCreatePlayer(player)
    player:display("Welcome in " .. ATME.getTheatre() .. " ... " .. player:getPseudo(), 10)
end
```

For other modules, activation in mission editor will be the same. If a module use an other module, all necessary modules have to be activated in the correct order.

In the following manual and examples, this integration and testing and will not be more exposed except in case of need. **Changes will be highlighted in green.**

Module 1 : Adding a dynamic display

This module takes up the base of the previous module but improve it by adding a displayed information regular updated.

Lua code

```
-- Advanced Tools for Mission Editor
local thisModule

local function onCreatePlayer(player)
    player:display("Welcome in " .. ATME.getTheatre() .. " ... " .. player:getPseudo(), 10)
end

local function onUpdatePlayer(player, events)
    local text = "Azimuth : " .. player:getAzimuth()

    if player:inAir() == true then
        player:display("In Air : " .. text, 5)
    else
        player:display("On floor : " .. text, 5)
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = onUpdatePlayer,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
```

```

onStopEngineAIUnitHandler = nil,

onCreateGroupHandler = nil,
onDeleteGroupHandler = nil,
onDisableGroupHandler = nil,

onCreateStaticObjectHandler = nil,
onDeleteStaticObjectHandler = nil,

onTimerHandler = nil,
onModuleStartHandler = nil,
}

thisModule = ATME.C_Module("Module1", newHandlers, true)
end

```

Explanations

- The module declares handler on **OnUpdatePlayer**. This callback will be called once every second for each active player; The player is passed in parameter, with **ATME.C_Player** instance. Any activated events will also be transmitted through the events parameter **ATME.C_EventsMgr** class instance. In this example, these events are not processed. The **OnUpdatePlayer** callback function creates a different display if the player is in the air or on the ground. It displays player azimuth at a T time (without taking into account the magnetic variation).

Tests

for Testing purpose, just follow the procedure described previously.



The azimuth may fluctuate a bit (don't move in your seat)

refresh every second.

Exercise

Exercise : add to display : MSL player altitude

Soluce :

```

local function onUpdatePlayer(player)
    local text = "Azimuth : " .. player:getAzimuth()
    text = text .. " - Alt MSL : " .. player:getMSLAltitude()

    if player:inAir() == true then
        player:display("In Air : " .. text, 5)
    else
        player:display("On floor : " .. text, 5)
    end
end
end

```

Module 1 : dynamic display replacement with a specific user Menu

This module takes up the base of the previous module but this time, the information will be displayed only on request of the player through the user menu F10.

Lua code

```

-- Advanced Tools for Mission Editor

local thisModule

local F10DisplayCommand
F10DisplayCommand = function(args)
    local player = args[1]

    local text = "Azimuth : " .. player:getAzimuth()
    text = text .. " - Alt MSL : " .. player:getMSLAltitude()

    if player:inAir() == true then
        player:display("In Air : " .. text, 5)
    else
        player:display("On floor : " .. text, 5)
    end
end

local function onCreatePlayer(player)
    player:display("Welcome in " .. ATME.getTheatre() .. " ... " .. player:getPseudo(), 10)

    local menu = player:getF10MenuRoot()
    menu:append(1, "Display informations", F10DisplayCommand, {player})
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
        onDeleteGroupHandler = nil,
    }
end

```

```

onDisableGroupHandler = nil,

onCreateStaticObjectHandler = nil,
onDeleteStaticObjectHandler = nil,

onTimerHandler = nil,
onModuleStartHandler = nil,
}

thisModule = ATME.C_Module("Module1", newHandlers, true)
end

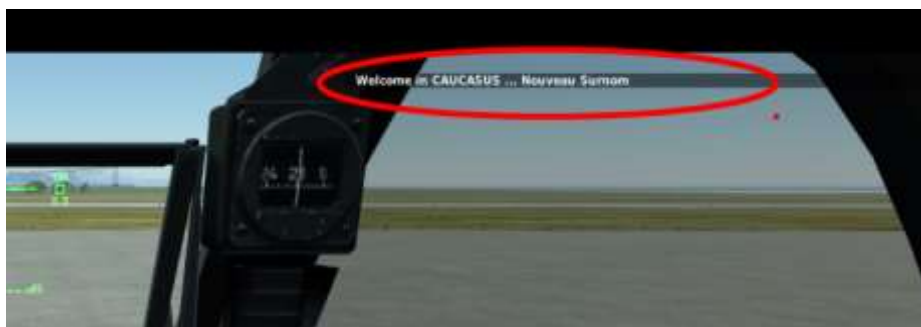
```

Explanation

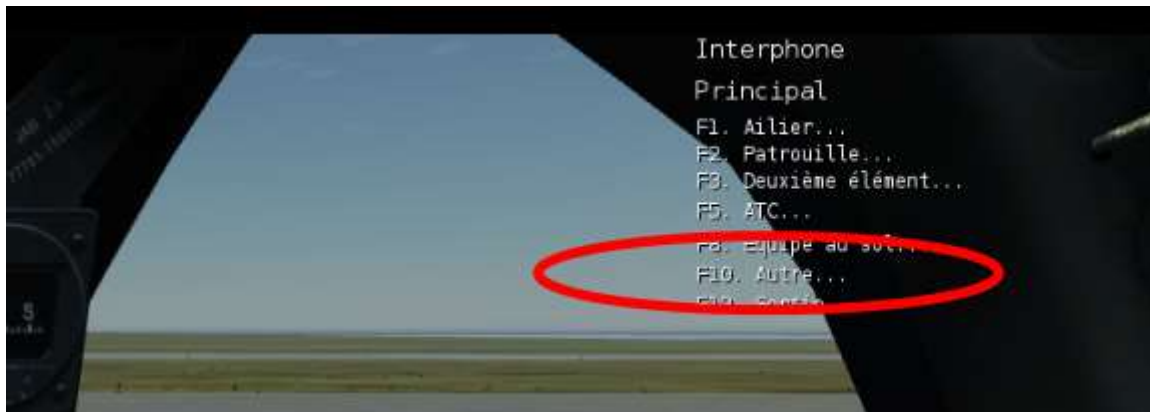
- A menu is always associated with its own function which will only be executed when the menu is used by the player (radio menu F10). This function, here **F10DisplayCommand**, has a somewhat special statement because it is local (lua limitation): the name is declared (local **F10DisplayCommand**) and then, at the next line the function itself. This function accepts a particular parameter **args** which is an array. This array is initialized in the **onCreatePlayer** section. **F10displayCommand** function returns display of the onUpdatePlayer callback from the previous example.
- The callback function **OnCreatePlayer** is supplemented with a menu variable which is initialized with an **ATME.C_F10Menu** class instance corresponding to the player base menu F10 thanks to the function player: **getF10MenuRoot**. In the following line, an entry is added to this menu by the menu function: **append** of the class **ATME.C_F10Menu** which admits:
 - A first parameter that will be explained later
 - second parameter is the label displayed in menu F10,
 - third is to the function **F10DisplayCommand** which will therefore be called when the choice of the entry of the menu F10 by the player will be made.
 - the last parameter are the parameters passed to the F10DisplayCommand function. (other variables if needed) In order to make the desired displays, the F10DisplayCommand function must know the instance of the player player.

Tests

This time, only the "Welcome" message display in a first attempt.



You must thactivate the user F10 Radio menu, (after F11 "back" if needed) :



select F10, th :



displays wanted informations :



Exercice

Exercice : Create a second choice entry in the menu F10 which will display Unit Player Name as defined in mission editor. associated function will be **F10DisplayNameCommand**

Soluce : creation of a new function (specific to this module) **F10DisplayNameCommand** and change the call-back onCreatePlayer function.

```
local F10DisplayNameCommand
```

```

F10displayCommand = function(args)
    local player = args[1]

    player:display("Unit player name : " .. player:getName(), 5)
end

local function onCreatePlayer(player)
    player:display("Welcome in " .. ATME.getTheatre() .. " ... " .. player:getPseudo(), 10)

    local menu = player:getF10MenuRoot()
    menu:append(1, "Display informations", F10DisplayCommand, {player})
    menu:append(1, "Display unit player name", F10DisplayNameCommand, {player})
end

```

- Adding menus entries (and submenus sebelow) will still the same as described, for all modules that will be created by users.

Module 1 : Adding a menu visible and available in some cases

This module takes up the base of the previous module but this time a new menu entry with the position in longitude / latitude is created and available only whthe player is on the ground.

This module will use the first argument of the function **menu: append** which allows to delete one or more menu items. Deletion is based on this value.

Lua code

```

-- Advanced Tools for Mission Editor

local thisModule

local F10DisplayCommand
F10DisplayCommand = function(args)
    local player = args[1]

    local text = "Azimuth : " .. player:getAzimuth()
    text = text .. " - Alt MSL : " .. player:getMSLAltitude()

    if player:inAir() == true then
        player:display("In Air : " .. text, 5)
    else
        player:display("On floor : " .. text, 5)
    end
end

local F10DisplayNameCommand
F10DisplayNameCommand = function(args)
    local player = args[1]

    player:display("Unit player name : " .. player:getName(), 5)
end

local F10DisplayPositionCommand
F10DisplayPositionCommand = function(args)
    local player = args[1]

    local position = player:getPosition()
    local lat, lon, alt = ATME.convertPointToLL(position)

    local dirlat, dlat, mlat, slat = ATME.convertToDMS(lat, false)
    local dirlon, dlon, mlon, slon = ATME.convertToDMS(lon, false)

    local westEast = "E"
    if dirlat == -1 then
        westEast = "W"
    end
end

```

```

end

local northSouth = "N"
if dirlon == -1 then
    northSouth = "S"
end

local text = string.format("Lat : %s%d°%d'%d\"\\n", westEast, dlat, mlat, slat)
text = text .. string.format("Lon : %s%d°%d'%d\"\\n", northSouth, dlon, mlon, slon)

player:display(text, 5)
end

local function onCreatePlayer(player)
    player:display("Welcome in " .. ATME.getTheatre() .. " ... " .. player:getPseudo(), 10)

    local menu = player:getF10MenuRoot()
    menu:append(1, "Display informations", F10DisplayCommand , {player})
    menu:append(1, "Display unit player name", F10DisplayNameCommand , {player})

    if player:inAir() == false then
        menu:append(2, "Display position", F10DisplayPositionCommand , {player})
    end
end

local function onTakeOffPlayer(player)
    local menu = player:getF10MenuRoot()

    menu:remove(2)
end

local function onLandingPlayer(player)
    local menu = player:getF10MenuRoot()

    menu:append(2, "Display position", F10DisplayPositionCommand , {player})
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = onTakeOffPlayer,
        onLandingPlayerHandler = onLandingPlayer,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = nil,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module("Module1", newHandlers, true)
end

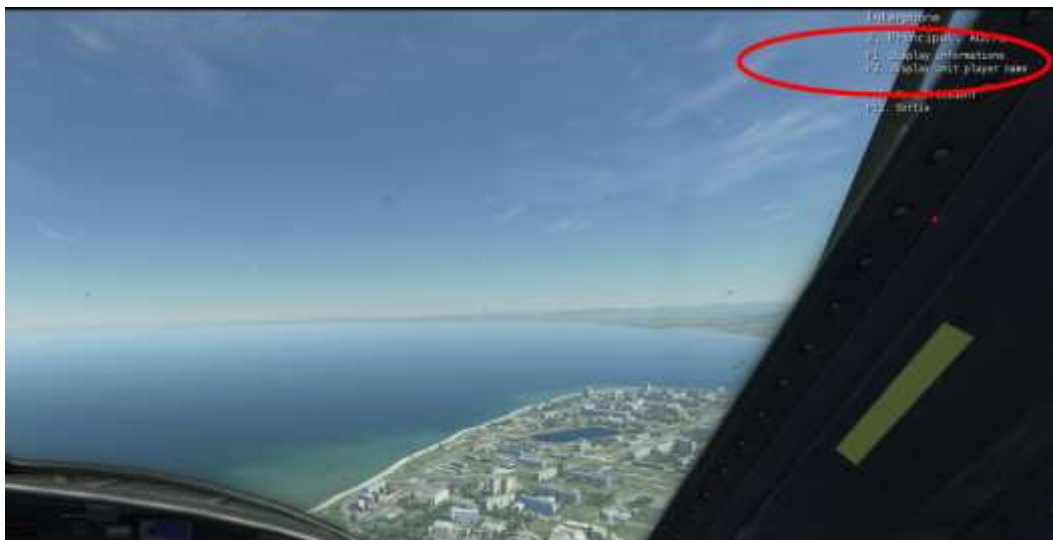
```

Explanations

- new module specific menu function **F10DisplayPositionCommand** is added. This function is used to convert a player position into displayable coordinates :
 - **player:getPosition** : to retrieve player position (point x y z)
 - **ATME.convertPointToLL** converts the point to latitude / longitude / altitude
 - **ATME.convertToDMS** transforms decimal degrees into minutes seconds completed from the direction.
 - **string.format** is a native lua function to format the text display.
- To manage take-off and landing, use the two callback **onTakeOffPlayer** and **onLandingPlayer**:
 - **onTakeOffPlayer** removes the menu entry whthe player takes off, with the function **menu: remove** which uses the **groupID** in parameter (value 2).
 - **onLandingPlayer** creates a new menu entry for displaying the ground position. The first parameter **menu: append** is 2 and is **groupID**. this menu is different from the two other because it will be deleted according to the context (player on the ground / player in the air)
- **onCreatePlayer** is modified to add a treatment if the player starts on the ground. Indeed, in this case, the position menu must exist from the beginning.

Tests

This time, F10 menus change. In air, position request doesn't display :



On Ground, position request is displayed :



And the request result is :



Exercise

Exercise : delete the two other menus at the first take-off. They will no longer be available.

Soluce :

```
local function onLandingPlayer(player)
    local menu = player:getFl0MenuRoot()
    menu:remove(1)
    menu:remove(2)
end
```

ou

```
local function onLandingPlayer(player)
    local menu = player:getFl0MenuRoot()
    menu:removeAll()
end
```

First case delete all entries with a **groupId** 1 and 2 .

Second case delete all entries.

Module 1 : Add a submenu and a player specific variable

This module use the base of the previous module. The two fixed inputs for displaying azimuth and altitude will be grouped into a submenu. To manage the submenu, a submenu variable will be added to the players. To do this, we will use :

`player.modules["Module1"].submenu`

"Module1" is the name of the module. It is possible to add as many variables as desired.

Lua code

```
-- Advanced Tools for Mission Editor

local thisModule

local Fl0DisplayCommand
Fl0DisplayCommand = function(args)
    local player = args[1]

    local text = "Azimuth : " .. player:getAzimuth()
    text = text .. " - Alt MSL : " .. player:getMSLAltitude()

    if player:inAir() == true then
        player:display("In Air : " .. text, 5)
    else
        player:display("On floor : " .. text, 5)
    end
end

local Fl0DisplayNameCommand
Fl0DisplayNameCommand = function(args)
    local player = args[1]

    player:display("Unit player name : " .. player:getName(), 5)
end

local Fl0DisplayPositionCommand
Fl0DisplayPositionCommand = function(args)
    local player = args[1]

    local position = player:getPosition()
    local lat, lon, alt = ATME.convertPointToLL(position)

    local dirlat, dlat, mlat, slat = ATME.convertToDMS(lat, false)
    local dirlon, dlon, mlon, slon = ATME.convertToDMS(lon, false)

    local westEast = "E"
    if dirlat == -1 then
        westEast = "W"
    end

    local northSouth = "N"
    if dirlon == -1 then
        northSouth = "S"
    end

    local text = string.format("Lat : %s%d°%d'%d\"\\n", westEast, dlat, mlat, slat)
    text = text .. string.format("Lon : %s%d°%d'%d\"\\n", northSouth, dlon, mlon, slon)

    player:display(text, 5)
end
```

```

local function onCreatePlayer(player)
    player:display("Welcome in " .. ATME.getTheatre() .. " ... " .. player:getPseudo(), 10)

    player.modules["Module1"].submenu = ATME.C_F10Menu(player, "General informations", nil)
    player.modules["Module1"].submenu:append(1, "Display informations",
                                              F10DisplayCommand , {player})
    player.modules["Module1"].submenu:append(1, "Display unit player name",
                                              F10DisplayNameCommand , {player})

    if player:inAir() == false then
        local menu = player:getF10MenuRoot()
        menu:append(2, "Display position", F10DisplayPositionCommand , {player})
    end
end

local function onTakeOffPlayer(player)
    local menu = player:getF10MenuRoot()
    menu:remove(2)
end

local function onLandingPlayer(player)
    local menu = player:getF10MenuRoot()
    menu:append(2, "Display position", F10DisplayPositionCommand , {player})
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = onTakeOffPlayer,
        onLandingPlayerHandler = onLandingPlayer,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = nil,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module("Module1", newHandlers, true)
end

```

Explanations

- **onCreatePlayer :**

- Create a submenu variable players relative. This variable is initialized with a submenu instance created by the class constructor **ATME.C_F10Menu**. Label "General information" will be displayed. The last parameter to **nil** defines where the submenu is created. As in our case, the submenu is created directly in the basic menu (root), **nil** is sufficient. If the menu has be created in another sub-menu (called parent menu), it would have been necessary to pass a parent variable in parameter.

- Two entries are created in the submenu. The syntax seems complex, in fact it is the created submenu variable followed by : **append** already used previously.
- The menu variable needed for basic menu to add the entry position is moved in the test because it is used only there.

Tests

This time a submenu in main menu is created called: « General informations ... »



This submenu has 2 entries :



Exercise

Exercise : delete the two other menus on take off. it won't be available anymore.

Soluce :

```
local function onTakeOffPlayer(player)
    local menu = player:getFl0MenuRoot()
    menu:remove(1)
    menu:remove(2)
end
```

or

```
local function onTakeOffPlayer(player)
    local menu = player:getFl0MenuRoot()
    menu:removeAll()
end
```

There is no change from the previous example. At runtime, and after take off, the submenu will disappear from menu F10, because there will be no more associated item.

Module 2 : DCS Interactions and User Trigger Management

Up to now, displays have been made either regularly, or through the use of F10 menus or submenus. This module addresses user triggers (User Trigger). For reasons of understanding, a new module is created, module "Module2". The mission has to include a ground unit called "Unit to explode". Otherwise, nothing will happen.

The aim of this module is to explode a unit either by the user menu or a 50 second end. As soon as the explosion occurs, entry F10 is deleted for all players.

We will use the `atme` function `ATME.C_AIUnit.getByname` which permits to retrieve a Unit by its name. If unit doesn't exist this function returns `nil`. `ATME.C_Player`, `ATME.C_Group`, `ATME.StaticObject` and `ATME.C_Race` classes have also this function available.

A particular focus has been used : The menu no longer appears to the players after the destruction of the unit, even if the player enters the mission.

Lua code

```
local thisModule
local moduleName = "Module2"

local unitIsDestroyed = false

local function explodeUnit()
    local unit = ATME.C_AIUnit.getByname("Unit to explode")

    if unit ~= nil then
        unit:explode(100)
        unitIsDestroyed = true
    end
end

local F10ExplodeCommand
F10ExplodeCommand = function()
    thisModule:removeUserTrigger("TEST1")
    explodeUnit()
    ATME.displayForAll("Unité détruite", 5)
end

local function onCreatePlayer(player)
    player:display("Welcome in " .. moduleName, 10)

    if unitIsDestroyed == false then
        local menu = player:getF10MenuRoot()
        menu:append(1, "Explode unit", F10ExplodeCommand, nil)
        player.modules["Module2"].menuOK = true
        if thisModule:userTriggerExists("TEST1") == false then
            thisModule:createAbsoluteUserTrigger("TEST1", "00:00:50")
        end
    else
        player.modules["Module2"].menuOK = false
    end
end

local function onUpdatePlayer(player, events)
    if unitIsDestroyed == true and player.modules["Module2"].menuOK == true then
```

```

        local menu = player:getF10MenuRoot()
        menu:remove(1)
        player.modules["Module2"].menuOK = false
    end
end

local function onTimer(events)
    if events.isUserTriggerEvent("TEST1") == true then
        explodeUnit()
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = onUpdatePlayer,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module(moduleName, newHandlers, true)
end

```

Explanations

- **explodeUnit** : This function is called by the local created function **F10ExplodeCommand** menu associated by a user trigger event in the callback function **onTimer**. it retrieves unit called « Unit to explode » with **ATME.C_AIUnit.getByname** because a AI unit is an instance from **ATME.C_AIUnit** class. If Unit exists it is destroyed by the explosion. The global variable **unitIsDestroyed** which was initially at **false** value move to **true** value on explosion event. This variable is used to manage the player radiomenu.
- **F10ExplodeCommand** : function called by the players menu entry F10. After calling **explodeUnit**, this function removes the user trigger called **TEST1** which is no longer needed.
- **onCreatePlayer** performs the following processing only if unit is not destroyed:
 - Create F10 players menu entry with **menu:append**
 - Assign true value to player variable **player.modules["Module2"].menuOK**. This variable is useful to avoid assigning multiple times in the process associated with

removing menus onUpdatePlayer. Even if this does not cause problems, it could slow down menus in case of large menus.

- Creating user trigger (if it does not exist) called **TEST1** and whose trigger is fixed at 50s after beginning of the mission by the function
thisModule:createAbsoluteUserTrigger("TEST1", "00:00:50").
- **onUpdatePlayer** is used to check every second if the unit is destroyed, and if yes, to delete the player menu entry of the assigned parameter. Once deleted, the variable **player.modules["Module2"].menuOK** is assigned to false value for each players. As a reminder, **onUpdatePlayer** is called once per second for each active player.
- **onTimer** will allow to trigger the explosion by controlling the event triggering with the user trigger **TEST1**.

Tests

Menu appears on Mission start.



If activated :



menu disappears :



If nothing is done before 50 seconds, the unit will explode without intervention and the menu will disappear in the same way.

Exercise

Exercise : Create a second user trigger named TEST2 which will trigger 20s after the Unit explosion will display to all players "End of Mission". Then, try naming it TEST1 and verify that it works. Why ??

Soluce :

```
local function explodeUnit()
    local unit = ATME.C_AIUnit.getByname("Unit to explode")

    if unit ~= nil then
        unit:explode(100)
        unitIsDestroyed = true
        thisModule:createAbsoluteUserTrigger("TEST2", "+00:00:20")
    end
end
```

The + before the delay indicates a delay beginning from the creation of the user trigger.

```
local function onTimer(events)
    if events.isUserTriggerEvent("TEST1") == true then
        explodeUnit()
    elseif events.isUserTriggerEvent("TEST2") == true then
        ATME.displayForAll("End of mission", 10)
    end
end
```

Answer : Although it still possible to create a new user trigger called TEST1, (which has previously beactivated or deleted), the processing of the two triggers is different in the onTimer function. They can not therefore have the same name.

Module 2 : User Triggers management with a Flag

Previous exemple is used, but user trigger **TEST1** will trigger 20s after activation of flag 1 in the mission editor flag armed after 10s. Also, unit will now explode one minute after the start of the mission.

```

local thisModule
local moduleName = "Module2"

local unitIsDestroyed = false

local function explodeUnit()
    local unit = ATME.C_AIUnit.getByName("Unit to explode")

    if unit ~= nil then
        unit:explode(100)
        unitIsDestroyed = true
    end
end

local F10ExplodeCommand
F10ExplodeCommand = function()
    thisModule:removeUserTrigger("TEST1")
    explodeUnit()
    ATME.displayForAll("Unité détruite", 5)
end

local function onCreatePlayer(player)
    player:display("Welcome in " .. moduleName, 10)

    if unitIsDestroyed == false then
        local menu = player:getF10MenuRoot()
        menu:append(1, "Explode unit", F10ExplodeCommand, nil)
        player.modules["Module2"].menuOK = true
        if thisModule:userTriggerExists("TEST1") == false then
            thisModule:createFlagRelativeUserTrigger("TEST1", 1, "00:00:50")
        end
    else
        player.modules["Module2"].menuOK = false
    end
end

local function onUpdatePlayer(player, events)
    if unitIsDestroyed == true and player.modules["Module2"].menuOK == true then
        local menu = player:getF10MenuRoot()
        menu:remove(1)
        player.modules["Module2"].menuOK = false
    end
end

local function onTimer(events)
    if events.isUserTriggerEvent("TEST1") == true then
        explodeUnit()
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = onUpdatePlayer,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
    }
end

```

```

onDeleteGroupHandler = nil,
onDisableGroupHandler = nil,

onCreateStaticObjectHandler = nil,
onDeleteStaticObjectHandler = nil,

onTimerHandler = onTimer,
onModuleStartHandler = nil,
}

thisModule = ATME.C_Module(moduleName, newHandlers, true)
end

```

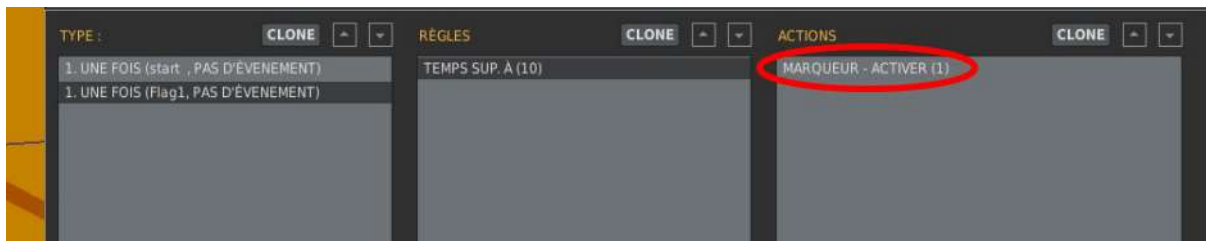
Explanations

- **onCreatePlayer** : the function **thisModule:createFlagRelativeUserTrigger** allows to create a user trigger depending of the flag activation. This flag can be armed in the mission editor or by the **ATME.setFlag** function. In this case, the second parameter is set to 1, indicating that user trigger will trigger 50 seconds after flag 1 is activated.

Tests

Initially, flag 1 is not activated. Nothing will happen if no player action is done.

In a second step, flag 1 is activated by adding a trigger in the mission editor:



Therefore, without player action, unit will explode after 60 seconds (50 seconds related to the ATME user trigger and 10 seconds related to the activation time of the flag).

Exercise

Exercise : Arm the flag 2 during the explosion and proceed to the display "End of mission" in the mission editor, 20s after this flag is activated.

Soluce :

```

local function explodeUnit()
    local unit = ATME.C_AIUnit.getByName("Unit to explode")

```

```

    if unit ~= nil then
        unit:explode(100)
        unitIsDestroyed = true
        ATME.setFlag(2, true)
    end
end

```

and

```

local function onTimer(events)
    if events.isUserTriggerEvent("TEST1") == true then
        explodeUnit()
    end
end

```

The user trigger TEST2 becomes not needed.

finalize by creating the needed DCS triggers in the mission editor.

Module 2 : User triggers with time Management

Previous exemple is used, but this time, if Unit is not destroy before by a, it will launch a flare per second during 10 seconds, after 20 seconds from mission beginning.

Lua code

```

local thisModule
local moduleName = "Module2"

local unitIsDestroyed = false

local function explodeUnit()
    local unit = ATME.C_AIUnit.getByName("Unit to explode")

    if unit ~= nil then
        unit:explode(100)
        unitIsDestroyed = true
    end
end

local F10ExplodeCommand
F10ExplodeCommand = function()
    thisModule:removeUserTrigger("TEST1")
    explodeUnit()
    ATME.displayForAll("Unité détruite", 5)
end

local function onCreatePlayer(player)
    player:display("Welcome in " .. moduleName, 10)

    if unitIsDestroyed == false then
        local menu = player:getF10MenuRoot()
        menu:append(1, "Explode unit", F10ExplodeCommand, nil)
        player.modules["Module2"].menuOK = true
        if thisModule:userTriggerExists("TEST1") == false then
            thisModule:createAbsoluteUserTrigger("TEST1", "00:00:50")
            thisModule:createAbsoluteUserTrigger("TEST2", "00:00:20 +10")
        end
    else
        player.modules["Module2"].menuOK = false
    end
end

local function onUpdatePlayer(player, events)

```

```

    if unitIsDestroyed == true and player.modules["Module2"].menuOK == true then
        local menu = player:getF10MenuRoot()
        menu:remove(1)
        player.modules["Module2"].menuOK = false
    end
end

local function onTimer(events)
    if events.isUserTriggerEvent("TEST1") == true then
        explodeUnit()
    elseif events.isUserTriggerEvent("TEST2") == true then
        local unit = ATME.C_AIUnit.getByName("Unit to explode")

        if unit ~= nil then
            unit:fireFlare("RANDOM")
        end
    end
end

end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = onUpdatePlayer,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module(moduleName, newHandlers, true)
end

```

Explanations

- **onCreatePlayer** : second paramater from function **thisModule:createAbsoluteUserTrigger** is now set to **"00:00:20 +10"** which define a start time at 20 seconds from the mission beginning, and a stop time 10 second after. Event associated with user trigger TEST2 will be repeated during 10 seconds.
- **onTimer** will launch a randomized color during all the activation time of user trigger **TEST2**

Tests

Nothing special here, the unit will fire its flares during the indicated time:



Exercise

Exercise : Without changing the user trigger, make sure that only one flare is sent, in other words, **unit:fireFlare** function should only be called once whthe user trigger is changed.

Soluce :

```
local function onTimer(events)
  if events.isUserTriggerEvent("TEST1") == true then
    explodeUnit()
  elseif events.isUserTriggerEventToggle("TEST2") == true then
    local unit = ATME.C_AIUnit.getByName("Unit to explode")

    if unit ~= nil then
      unit:fireFlare("RANDOM")
    end
  end
end
```

isUserTriggerEventToggle will only activate once evif the UserTrigger has a defined duration.

Module 3 : Mission data for groups and dynamic group creation

In this module, units groups will be created from groups defined in the mission. In ATME, it is the only way to create a group of units dynamically. On the other hand, it is possible to assign some or all waypoints to another group of the mission. It is also possible to dynamically create a group from a group whose activation is delayed in the mission editor.

Currently, only units groups on the ground can be dynamically duplicated.

ATME_C_GroupSpawnDatas class allows duplication management. Two functions are required:

- **ATME.C_GroupSpawnDatas.duplicateFromMissionDatas** returns an instance of the **ATME.C_GroupSpawnDatas** class and retrieves the data from a defined group in the mission and gives it a new name. Names of the associated units will also be changed.
- **datas:spawn** allows dynamic creation of the new group. If group already exists, an error will be generated. **Datas** is the instance returned by the previous function in case if no error is returned.

It is possible to reposition the new group and allocate to it all or some of the waypoints from another group also defined in the mission.

This module attempts to duplicate a group called "origin" into a group called "copy0". The new group will be randomly spawned in a DCS area defined in the mission editor and called "spawnZone". Spawn is done as soon as the mission is started on callback **onStart**

Lua code

```
local thisModule
local moduleName = "Module3"

local function onStart(areUnitsAndPlayersInitialized)
    if areUnitsAndPlayersInitialized == true then
        local err, var1 = ATME.C_GroupSpawnDatas.duplicateFromMissionDatas("origine", "copy0")
        if err == false then
            var1:spawn("spawnZone")
        end
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = nil,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
```

```
onLandingAIUnitHandler = nil,  
onStartEngineAIUnitHandler = nil,  
onStopEngineAIUnitHandler = nil,  
  
onCreateGroupHandler = nil,  
onDeleteGroupHandler = nil,  
onDisableGroupHandler = nil,  
  
onCreateStaticObjectHandler = nil,  
onDeleteStaticObjectHandler = nil,  
  
onTimerHandler = nil,  
onModuleStartHandler = onStart,  
}  
  
thisModule = ATME.C_Module(moduleName, newHandlers, true)  
end
```

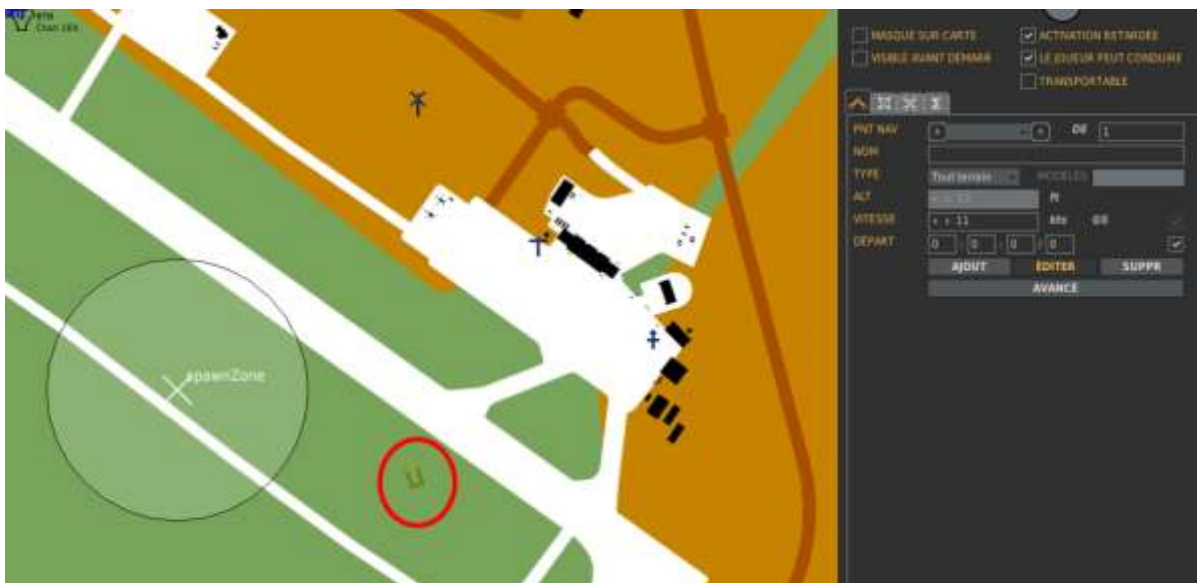
Explanations

- **onStart**: This module is called twice during startup. if **areUnitsAndPlayersInitialized** parameter is set to **true**, this is the second call after the units initialization defined in the mission. If the copy of the data happens without error, the group will be duplicated somewhere in the DCS zone **spawnZone**.

It is of course possible to combine the functions presented above to arrive at a more complex scenario.

Tests

group is created with a delayed activation.



That mission launch, the new group is present in the right zone.



Exercise

Exercise : In the mission editor, associate at least 4 waypoints to the group called "origin". Add a ground unit called "reference" near the 3rd waypoint. Then check " activate later" so "origin" group no longer appears. In lua, dynamically create a new spawned group called "copy1" after 10s, in the DCS "spawnZone" zone and associate it with all of the waypoints from the "origin" group from the waypoint closest to the created unit.

Soluce :

```
local function onTimer(events)
    if events.isUserTriggerEvent("TEST1") == true then
        local unit = ATME.C_AIUnit.getByName("reference")
        if unit ~= nil then
            local err, var1 = ATME.C_GroupSpawnDatas.duplicateFromMissionDatas("origine",
                                                                                "copy1")

            if err == false then
                var1:spawn("spawnZone", "origine", unit)
```

```

        end
    end
end

local function onStart(areUnitsAndPlayersInitialized)
    if areUnitsAndPlayersInitialized == true then
        local err, var1 = ATME.C_GroupSpawnDatas.duplicateFromMissionDatas("origine", "copy0")
        if err == false then
            var1:spawn("spawnZone")
        end
    end

    thisModule:createAbsoluteUserTrigger("TEST1", "00:00:10")
end
end

```

We create a user trigger that will trigger through the callback function **onTimer**. **var1:spawn** function is used to directly retrieve all waypoints from the "origin" group from the waypoint closest to the "reference" unit.

```

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = nil,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = onStart,
    }

    thisModule = ATME.C_Module(moduleName, newHandlers, true)
end

```

onTimer is now used

Module 4 : Core Events management

ATME manages **ATME Core events** such as counters for races, signals or infantry boarding in a transport unit.

Whthese events are triggered, ATME informe all active modules using the **ATME.C_EventMgr** class through two callback functions **onUpdatePlayerHandler** and **onTimerHandler**. See the **ATME.C_Module** class for details. The details of the Core Event types in ATME are listed in the **ATME.C_EventMgr** class.

This mission is to report a vehicle (whose group is called "My Vehicle") as soon as a friendly helicopter flies over the defined detection area. As with any automatic reporting, this helicopter

must have troop carrying capacity. This will be indicated by a gresmoke and a message will be displayed. The list of authorized to transport troops helicopters is givin the **load** function of the **ATME.C_Player** or **ATME.C_AIUnit** classes.

In DCS, the smoke is present for 5 minutes. If the helicopter flies over the area beyond this time, the group will automatically pop up a new smoke within 10 to 90 seconds.

Lua code

```
-- Advanced Tools for Mission Editor
local thisModule
local moduleName = "Module4"

local function onTimer(events)
    for _id, _ in events:pairs() do
        if events:isCoreEvent(_id) == true then
            if events:getCoreEventType(_id) == "SIGNAL_UNIT_IN_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display("Vous entrez dans la zone", 5)
            elseif events:getCoreEventType(_id) == "SIGNAL_UNIT_OUT_OF_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display("Vous sortez de la zone", 5)
            end
        end
    end
end

local function onCreateGroup(_group)
    if _group:getName() == "My Vehicle" then
        _group:setSignal(2500, "SIGNAL_SMOKE GREEN")
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = nil,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = onCreateGroup,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module(moduleName, newHandlers, true)
end
```

Explanations

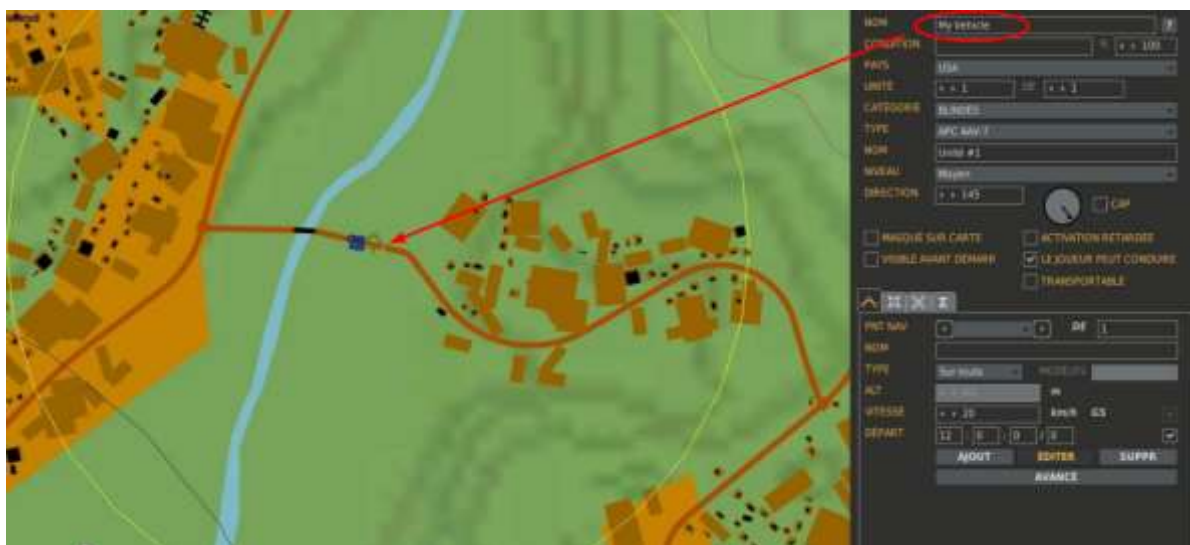
- **onCreateGroup**: The group associated with the vehicle activates an automatic signal over an area of radius 3km. The report will be a gresmoke. It is also possible to signal themselves by flares.
- **onTimer** will process core events with a **for loop** and the **isCoreEvent** function. a test on the type allows to know the emitted Core event. It is thpossible to recover all the specific data from the event and especially the player in the present case (**data.unit** is the unit entering or leaving the zone).

Tests

units are created on Sochi for the helicopter and nearby for transport and the infantry group.



troop transport vehicle is created:



Infantry units are only used for the following practice:



Exercise

Exercise : In mission editor, add an infantry group called "rescue1" nearby (<200m) from the vehicle created. Check that this vehicle has troop carrying capacity. As soon as the helicopter lands nearby (<500m), infantry group will board immediately. If at least one helicopter enters the zone and leaves without any other being in hover flight, the group will then board the vehicle. Inform player as soon as the group has boarded (in all cases no matter boarding in the helicopter or in the vehicle).

Useful: To be able to embark, an infantry group must be ready to embark. This is done by the **changeReadyToBoard(true)** from **ATME.C_Group** class.

Soluce :

```

local nbHelicoptersInZone = 0
local noMoreHelicopterInZone = false

local function onTimer(events)
    local group = ATME.C_Group.getByName("rescue1")

    for _id, _ in events:pairs() do
        if events.isCoreEvent(_id) == true then
            if events:getCoreEventType(_id) == "SIGNAL_UNIT_IN_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display("Vous entrez dans la zone", 5)
                nbHelicoptersInZone = nbHelicoptersInZone + 1
            elseif events:getCoreEventType(_id) == "SIGNAL_UNIT_OUT_OF_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display("Vous sortez de la zone", 5)
                nbHelicoptersInZone = nbHelicoptersInZone - 1
                if nbHelicoptersInZone == 0 and group ~= nil then
                    noMoreHelicopterInZone = true
                end
            end
        end
    end

    if noMoreHelicopterInZone == true then
        local unit = ATME.C_Group.getByName("My Vehicle"):getFirstUnit()
    end
end

```

```

        unit:load(group, 200)
    end
end

local function onLanding(player)
    local group = ATME.C_Group.getByname("rescue1")
    if group ~= nil then
        local err, errName = player:load(group, 500)

        if err == true then
            if errName == "LOAD_UNIT_TOO_FAR" then
                player:display(displayLabels[ATME.getLanguage()][4], 5)
            else
                player:display("Erreur ... " .. errName, 5)
            end
        end
    end
end

local function onCreateGroup(_group)
    if _group:getname() == "My Vehicle" then
        _group:setSignal(2500, "SIGNAL SMOKE GREEN")
    elseif _group:getname() == "rescue1" then
        _group:changeReadyToBoard(true)
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = nil,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = onLanding,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = onCreateGroup,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module(moduleName, newHandlers, true)
end

```

nbHelicoptersInZone et **noMoreHelicopterInZone** variables are created and initialized to manage helicopters if there are multiple (multiplayer). As soon as a player enters the zone, **nbHelicoptersInZone** increase by 1. As soon as an helicopter leaves the zone, nbHelicoptersInZone drops by 1. As soon as there is no more helicopter in zone **noMoreHelicopterInZone** flag is set to true if if no one has recovered the infantry group. In this case, it is the vehicle that ships the group (radius of 200m for loading, if it is positioned beyond, nothing will happen).

If an helicopter lands nearby (onLanding, max loading distance 500m), and the group has not yet beembarked, it is thembarked if the speed is less than 3m / s, otherwise an error will be displayed. If the helicopter is too far, a specific message is displayed.

Whcreating the group "rescue1" it is set waiting for boarding with **changeReadyToBoard**.

complement : Add a message to all players as soon as the group is embarked.

```
local function onTimer(events)
    local group = ATME.C_Group.getByname("rescue1")

    for _id, _ in events:pairs() do
        if events:isCoreEvent(_id) == true then
            if events:getCoreEventType(_id) == "SIGNAL_UNIT_IN_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display("Vous entrez dans la zone", 5)
                nbHelicoptersInZone = nbHelicoptersInZone + 1
            elseif events:getCoreEventType(_id) == "SIGNAL_UNIT_OUT_OF_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display("Vous sortez de la zone", 5)
                nbHelicoptersInZone = nbHelicoptersInZone - 1
                if nbHelicoptersInZone == 0 and group ~= nil then
                    noMoreHelicopterInZone = true
                end
            elseif events:getCoreEventType(_id) == "TRANSPORT_END_OF_BOARDING" then
                ATME.displayForAll("Le groupe d'infanterie a été embarqué", 5)
            end
        end
    end

    if noMoreHelicopterInZone == true then
        local unit = ATME.C_Group.getByname("My Vehicle"):getFirstUnit()
        unit:load(group, 200)
    end
end
```

Module 4 : pickupzones Management

pickupzones are DCS zones where infantry groups are waiting to board. Infantry groups must be present at a givtime in the area. Also the path must pass through the zone.

These zones can be reported through infantry groups using the **setSignal** function from **ATME.C_Group class**. In this case, if several groups have this possibility, only one will deal with reporting to the helicopters coalition. If it is destroyed, another will take over if the latter has also signaled .

Lua code

```
-- Advanced Tools for Mission Editor
local thisModule
local moduleName = "Module4"

local function onTimer(events)
    for _id, _ in events:pairs() do
        if events:isCoreEvent(_id) == true then
            if events:getCoreEventType(_id) == "SIGNAL_UNIT_IN_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display("Vous entrez dans la zone", 5)
            end
        end
    end
end
```

```

end

local function onCreateGroup(_group)
    if _group:getName() == "rescue1" or _group:getName() == "rescue2" then
        _group:setPickupZone("ZonePickup1", 1)
        _group:setSignal(2500, "SIGNAL_SMOKE RED")
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = nil,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = onCreateGroup,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module(moduleName, newHandlers, true)
end

```

Explications

- **onCreateGroup**: there are two infantry groups "rescue1" and "rescue2". These two groups pass through the DCS zone "ZonePickup1" by construction in the mission editor. Once these groups are in zone, they stand themselves on hold. The smoke will only be activated when helicopter is in the hover fly zone defined by the setSignal function (here 2.5km).

Le reste est entièrement automatique. This test can be completed by the **ATME_Rescue module available for download**. This module makes it possible to manage the embarkation / disembarkation of infantry troops.

Tests

Two infantry groups were created near Sochi. helicopter is on Sochi airfield.

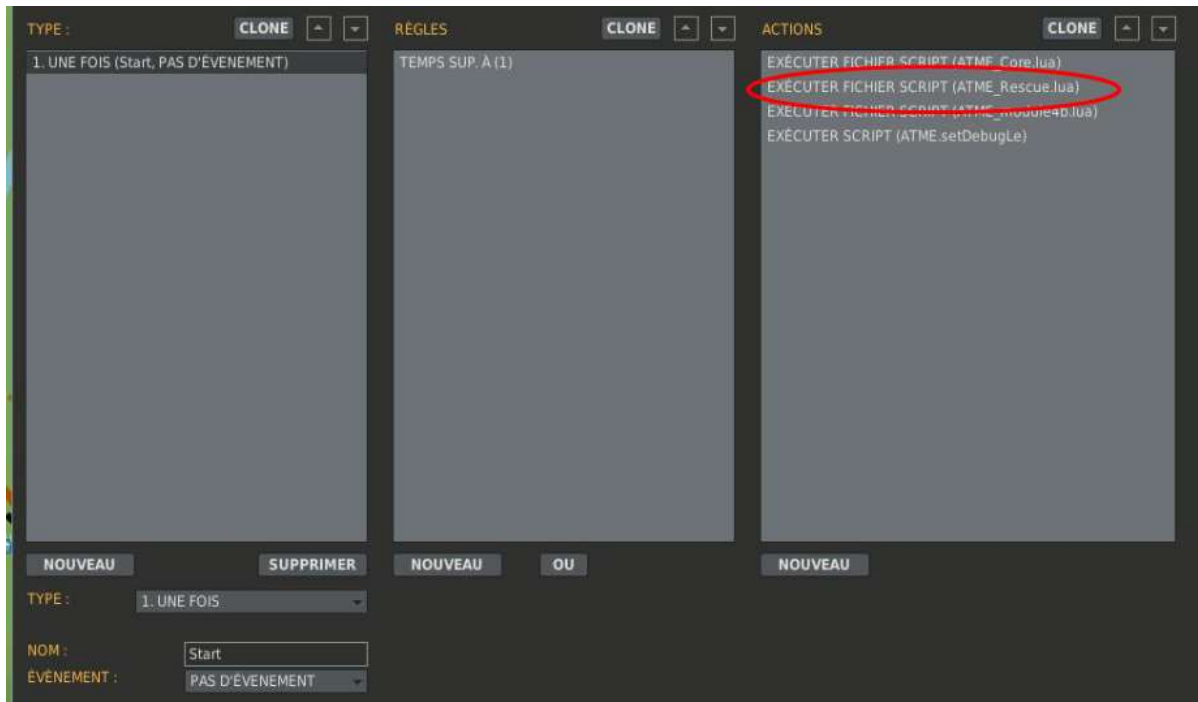


At mission start and before taking off, infantry groups will be observed in motion and positioning themselves standing on hold in the area.

Whthe helicopter enters the area, a red smoke (unique for both groups) pops.

Exercice

Exercice : This test can be completed by the ATME_Rescue module available for download. This module makes it possible to manage the embarkation / disembarkation of infantry troops. Download it and integrate it into the mission to bring back the infantry groups on Sochi with the helicopter..



Module 4 : Multilangage management

ATME supports multilangage management. Messages from Exercise "Module 4: Managing Core Events" will be translated into English. The module will thsupport English and French :

- Each message must be inserted in a table that itself has an entry for French, FR, and an entry for English, EN. This table will be a **local variable** of the module. It can be called **displayLabels**
- Messages with dynamic values will be inserted using lua's **string.format** function

Exemple :

```
local displayLabels = {
  FR = {
    "Transport de troupes",
    "Embarquer un groupe",
    "Faire débarquer %s",
    "Vous n'êtes pas posé !",
    "Vous êtes trop loin du groupe à secourir !",
    "Groupe %s embarqué !",
    "Groupe %s débarqué sans pertes !",
    "Groupe %s débarqué avec une perte !",
    "Groupe %s débarqué avec %d pertes !",
    "Il reste %d unités à bord.",
    "Il ne reste nonee unité à bord.",
    "Vous ne pouvez pas débarquer d'unité ici, trop de présence d'eau !",
    "Tout le groupe s'est noyé.",
    "None groupe à récupérer.",
    "Vous n'avez pas assez de place pour embarquer tout le groupe !"
  },
  = {
    "Personnel carrier",
```

```

        "Load group",
        "Unload %s",
        "You're in air !",
        "You're too far from troops to rescue !",
        "Group %s on board !",
        "Group %s disembarked without losses !",
        "Group %s disembarked with losses !",
        "Group %s disembarked with losses !",
        "There's %d units on board.",
        "No more unit on board.",
        "You can't disembark units here. Too much water !",
        "All group drowned.",
        "No group to rescue.",
        "You can't embark all units !"
    },
}

```

Lua code

```

-- Advanced Tools for Mission Editor
local thisModule
local moduleName = "Module4"

local nbHelicoptersInZone = 0
local noMoreHelicopterInZone = false

local displayLabels = {
    FR = {
        "Vous entrez dans la zone",
        "Vous sortez de la zone",
        "Le groupe d'infanterie a été embarqué",
        "Vous vous êtes posé trop loin",
    },
    = {
        "You enter in zone",
        "You exit the zone",
        "Infantry group is on board",
        "You land too far",
    },
}

local function onTimer(events)
    local group = ATME.C_Group.getByName("rescue1")

    for _id, _ in events:pairs() do
        if events.isCoreEvent(_id) == true then
            if events:getCoreEventType(_id) == "SIGNAL_UNIT_IN_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display(displayLabels[ATME.getLanguage()][1], 5)
                nbHelicoptersInZone = nbHelicoptersInZone + 1
            elseif events:getCoreEventType(_id) == "SIGNAL_UNIT_OUT_OF_ZONE" then
                local datas = events:getCoreEventDatas(_id)
                datas.unit:display(displayLabels[ATME.getLanguage()][2], 5)
                nbHelicoptersInZone = nbHelicoptersInZone - 1
                if nbHelicoptersInZone == 0 and group ~= nil then
                    noMoreHelicopterInZone = true
                end
            elseif events:getCoreEventType(_id) == "TRANSPORT_END_OF_BOARDING" then
                ATME.displayForAll(displayLabels[ATME.getLanguage()][3], 5)
            end
        end
    end

    if noMoreHelicopterInZone == true then
        local unit = ATME.C_Group.getByName("My Vehicle"):getFirstUnit()
        unit:load(group, 200)
    end
end

```

```

local function onLanding(player)
    local group = ATME.C_Group.getByName("rescue1")

    if group ~= nil then
        local err, errName = player:load(group, 500)

        if err == true then
            if errName == "LOAD_UNIT_TOO_FAR" then
                player:display(displayLabels[ATME.getLanguage()][4], 5)
            else
                player:display("Erreur ... " .. errName, 5)
            end
        end
    end
end

local function onCreateGroup(_group)
    if _group:getName() == "My Vehicle" then
        _group:setSignal(2500, "SIGNAL_SMOKE_GREEN")
    elseif _group:getName() == "rescue1" then
        _group:changeReadyToBoard(true)
    end
end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = nil,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = nil,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = onLanding,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onDisableAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = onCreateGroup,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = nil,
    }

    thisModule = ATME.C_Module(moduleName, newHandlers, true)
end

```

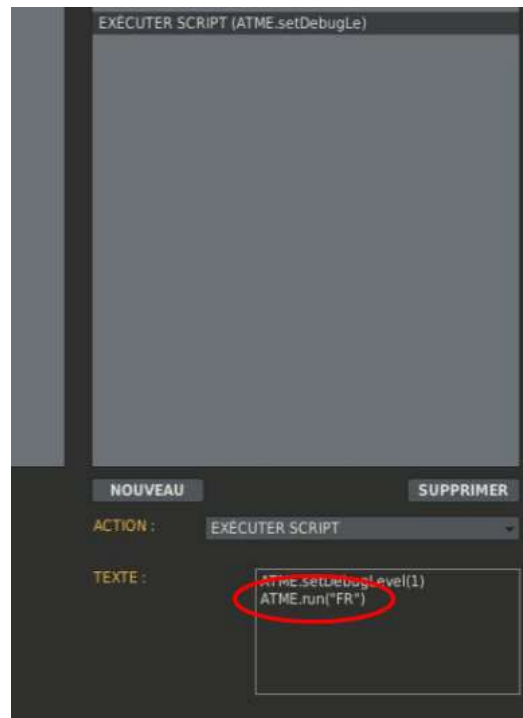
Explanations

Highlighted, the multilingual modification

- **displayLabels:** This table contains all texts in English and French. The translation order is essential, each message index for all languages must be identical
- Each display uses the correct label and uses the **ATME.getLanguage** function and the correct index.

Tests

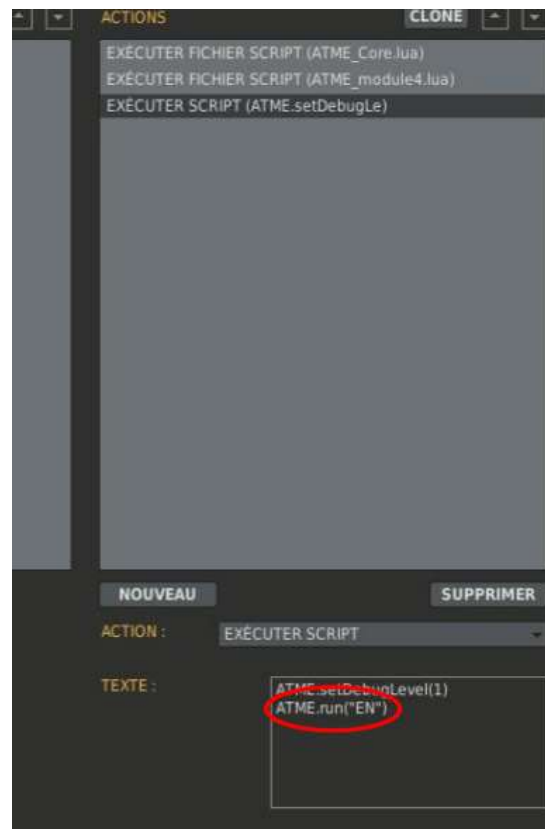
The choice of the language will thbe made in the mission editor, through the **ATME.run** function.



The chosen language is French and the display is:



The chosen language is English:



Module 5 : Race management

ATME offers the ability to create and manage races with measure of time and rankings.

This module manages a small race around Batumi with a few gates, a measure of intermediate and final time. The maximum distance between two sides of a door is 500m.

Lua Code

```
-- Advanced Tools for Mission Editor
local thisModule
local moduleName = "Module5"

local function onCreatePlayer(player)

    local race = ATME.C_Race.getByName("Ma course")

    race:addPlayer(player)
end

local function onTimer(events)
    for _id, _ in events:pairs() do
        if events.isCoreEvent(_id) == true then
            thisModule:output(events:getCoreEventType(_id), 1)
            local typeEvent = events:getCoreEventType(_id)
            local datas = events:getCoreEventDatas(_id)

            if ATME.isObjectClass(datas.unit, ATME.C_Player) == true then
                if typeEvent == "RACE_START" then
                    datas.unit:display("Départ ...", 5)
                elseif typeEvent == "RACE_TURN_TIME" then
                    local err, h, m, s, d = ATME.convertToHMSd(datas.playTime)
                    local text = string.format("%02d:%02d:%02d", m, s, d)
                    datas.unit:display("Fin du tour N°" .. datas.turnNumber ..
                                     " - Temps : " .. text, 5)

                elseif typeEvent == "RACE_BEST_TURN_TIME" then
                    local err, h, m, s, d = ATME.convertToHMSd(datas.playTime)
                    local text = string.format("%02d:%02d:%02d", m, s, d)
                    datas.race:displayForAllPlayers("Nouveau record du tour pour " ..
                                                    datas.unit:getPseudo() .. " : " .. text,
                                                    5)

                elseif typeEvent == "RACE_PLAYER_BEST_TURN_TIME" then
                    datas.unit:display("Votre meilleur tour", 5)

                elseif typeEvent == "RACE_LAP_TIME" then
                    local err, h, m, s, d = ATME.convertToHMSd(datas.playTime)
                    local text = string.format("%02d:%02d:%02d", m, s, d)
                    datas.unit:display("Porte N°" .. datas.doorNumber ..
                                     " - Temps : " .. text, 5)

                elseif typeEvent == "RACE_BEST_LAP_TIME" then
                    local err, h, m, s, d = ATME.convertToHMSd(datas.playTime)
                    local text = string.format("%02d:%02d:%02d", m, s, d)
                    datas.race:displayForAllPlayers("Meilleur tps intermédiaire Porte N° " ..
                                                    datas.doorNumber .. " pour " ..
                                                    datas.unit:getPseudo() .. " : " .. text,
                                                    5)

                elseif typeEvent == "RACE_BEST_FINAL_TIME" then
                    datas.race:displayForAllPlayers("Nouveau record pour " ..
                                                    datas.unit:getPseudo(), 5)
                end
            end
        end
    end
end
```

```

elseif typeEvent == "RACE_PLAYER_BEST_FINAL_TIME" then
    datas.unit:display("Vous battez votre record de course", 5)

elseif typeEvent == "RACE_FINAL_TIME" then
    local err, h, m, s, d = ATME.convertToHMSd(datas.totalPlayTime)
    local text = "Temps final pour " .. datas.unit:getPseudo() .. " : " ..
        string.format("%02d:%02d:%02d", m, s, d) .. "\n"
    local ranking = datas.race:getRanking()

    text = text .. "Classement général :\n"
    for _i, _item in ipairs(ranking) do
        local err, h, m, s, d = ATME.convertToHMSd(_item.bestTime)
        local timeText = "--:--:--"
        if err == false then
            timeText = string.format("%02d:%02d:%02d", m, s, d)
        end
        text = text .. string.format("%2d - ", _i) ..
            _item.player:getPseudo() .. " avec " .. timeText .. "\n"
    end

    datas.race:displayForAllPlayers(text, 5)

elseif typeEvent == "RACE_DOOR_MISSED" then
    datas.unit:display("Porte N°" .. datas.doorNumber ..
        " loupée : pénalité de 10s", 5)

elseif typeEvent == "RACE_PLAYER_TOO_HIGH" then
    datas.unit:display("Altitude trop élevée : pénalité de 10s", 5)
end
end
end
end
end

local function startOn(areUnitsAndPlayersInitialized)
    if areUnitsAndPlayersInitialized == false then
        local myRace = ATME.C_Race("Ma course", "PiquetG", "PiquetD", 2)

        myRace:setMaxAltitudeRule(200, 10)
        myRace:setMissedDoorRule(10)

        if myRace ~= nil then
            myRace:setLapAtDoor(4)
        end
    end
end

end

-- MAIN
do
    local newHandlers = {
        onCreatePlayerHandler = onCreatePlayer,
        onDeletePlayerHandler = nil,
        onUpdatePlayerHandler = onUpdatePlayer,
        onTakeoffPlayerHandler = nil,
        onLandingPlayerHandler = nil,
        onStartEnginePlayerHandler = nil,
        onStopEnginePlayerHandler = nil,

        onCreateAIUnitHandler = nil,
        onDeleteAIUnitHandler = nil,
        onTakeoffAIUnitHandler = nil,
        onLandingAIUnitHandler = nil,
        onStartEngineAIUnitHandler = nil,
        onStopEngineAIUnitHandler = nil,

        onCreateGroupHandler = nil,
        onDeleteGroupHandler = nil,
        onDisableGroupHandler = nil,

        onCreateStaticObjectHandler = nil,
        onDeleteStaticObjectHandler = nil,

        onTimerHandler = onTimer,
        onModuleStartHandler = startOn,
    }

```



```
thisModule = ATME.C_Module(moduleName, newHandlers, true)
end
```

Explanations

- **onCreatePlayer** will allow players to be associated with the race. The management is automatic. It is of course possible to unsubscribe a player at any time.
- **onStart** will create the "My race" name race. This race has two laps. Penalties are associated in the case of maximum height exceeded or door missed.
- **onTimer** will manage the race events and thus display the times or penalties.

Tests

In the mission editor, create two planes (warbirds) and race (2 pickets set side of doors):



The pickets shown above by rectangles are numbered from #001 to #006 (without stakes #003):

NOM	NPS	STRT	QTR
Player0000	USA	0	0
Player0001	USA	0	0
Player0002	USA	0	0
Player0003	USA	0	0
Player0004	USA	0	0
Player0005	USA	0	0
Player0006	USA	0	0
Player0007	USA	0	0
Player0008	USA	0	0
Player0009	USA	0	0
Player0010	USA	0	0

A left picket of a given number is associated with a right picket of the same given number to form a gate.

During the race, intermediate time:



First Lap :



End of Race and ranking (« classement » in french) :



If there's some player, a ligne of ranking will exist for all of them.

Global functions of the ATME table

ATME.arePointsVisible(pA, pB)

Description : check if two points are visible from each other, without taking into account buildings and any obstacles other than ground.

Parameters :

pA	table	3D Point x, y et z, ou 2D Point x,y
pB	table	3D Point x, y et z, ou 2D Point x,y

Return Values:

Without error :	boolean	True if the 2 point can " see each other" else false.
If error :	nil	

ATME.convertFtstoM(distance)

Description: converting in meters a feet distance

Parameters :

distance	number	Distance feet
-----------------	--------	---------------

Return values:

Without error :	number	Distance converted in feet
If error	nil	

ATME.convertKphtoMps(speed)

Description: convert in m/s a km/h speed

Parameters :

speed	number	Speed km/h
--------------	--------	------------

Return values:

Without error :	number	Converted speed m/s
If error	nil	

ATME.convertLLToMGRS(latitude, longitude)

Description: convert a latitude and longitude to a MGRS format table.

Parameters :

latitude	number	Latitude degrees
longitude	number	Longitude degrees

Return values:

Without error :	table	MGRS Table
If error	nil	

ATME.convertLLToPoint(latitude, longitude, altitude)

Description: allowing the conversion of latitude, longitude and altitude in 3D Point. (x y z)

Parameters :

latitude	number	Latitude degrees
longitude	number	Longitude degrees
altitude	number	Altitude above sea level m

Return values:

Without error :	table	3D Point x, y z
If error	nil	

ATME.convertMGRSToLL(tableMGRS)

Description : converting a correctly initialized MGRS table into latitude / longitude

Parameters :

tableMGRS	table	MGRS Table
------------------	-------	------------

Return values:

Without error :	number	Latitude degrees
	number	Longitude degrees
If error	nil	

ATME.convertMpstoKph(speed)

Description :

Function converting in km / h a m / s. speed

Parameters :

speed	number	speed m/s
--------------	--------	-----------

Return values:

Without error :	number	Converted speed km/h
If error	nil	

ATME.convertMpstoNds(speed)

Description : Function converting in knots a m / s speed

Parameters :

speed	number	speed m/s
--------------	--------	-----------

Return values:

Without error :	number	Knots converted speed
If error	nil	

ATME.convertMtoFts(distance)

Description : Function converting in feet a meter distance or altitude

Parameters :

distance	number	distance m
-----------------	--------	------------

Return values:

Without error :	number	Feet converted distance
If error	nil	

ATME.convertMtoNM(distance)

Description: Function converting in nautique miles a meter distance

Parameters :

distance	number	distance m
-----------------	--------	------------

Return values:

Without error :	number	Nautique miles distance
If error	nil	

ATME.convertNdstoMps(speed)

Description : Function converting in m/s a knots speed

Parameters :

speed	number	Knots speed
--------------	--------	-------------

Return values:

Without error :	number	m/s speed
If error	nil	

ATME.convertNMtoM(distance)

Description : Function converting in meters a nautique miles distance :

distance	number	nautique miles distance
-----------------	--------	-------------------------

Return values:

Without error :	number	Meter distance
If error	nil	

ATME.convertPointToLL(point)

Description :

Function to convert a 3D point into latitude, longitude and altitude.

Parameters :

point	table	3D Point x, y et z, ou 2D Point x,y
--------------	-------	-------------------------------------

Return values:

Without error :	number	Latitude degrees
	number	Longitude degrees
	number	Altitude above sea level in meter
If error	nil	

ATME.convertToDMS(angle, secDec)

Description : Function to convert into a decimal latitude or longitude (degrees, minutes, decimal minutes) or latitude or longitude in degrees, minutes, seconds.

Parameters :

angle	number	Angle à convertir (-180° à +180°)
secDec	boolean	Si true, conversion degrés, minute, secondes. Sinon conversion degrés, minutes et minutes décimales.

Return values:

Without error :	number	Direction 1 si angle >= 0, -1 sinon
	number	Degrés entiers valeur absolue
	number	Minutes entières valeur absolue
	number	Centièmes entiers de minutes ou secondes entières suivant secDec
If error	nil	

ATME.convertToHMSd(seconds)

Description : Function to convert seconds in hour, minutes, seconds and 1/100th of seconds.

Parameters :

seconds	number	Seconds to be converted
----------------	--------	-------------------------

Return values:

Without error :	boolean	False, seconds >= 0, true sinon dans ce dernier cas, pas d'autres retours
	number	Hour if no error
	number	Minutes if no error
	number	Seconds if no error
	number	1/100th of seconds
If error	nil	

ATME.convertToPoint3D(point)

Description : function converting a2D Point (x,y) into 3D (x,y,z). if point is already in 3D, function does nothing.

Parameters :

point	table	3D Point x, y et z, or 2D Point x,y
--------------	-------	-------------------------------------

Return values:

Without error :	table	3D Point x, y et z
If error	nil	

ATME.displayForAll(text, duration)

Description : This function displays a message to all mission players.

Parameters :

text	string	Text to display
duration	number	Duration to display the message in seconds

Return values: Nonee

ATME.displayForCoalition(coalitionName, text, duration)

Description : This function displays a message to specific coalition players.

Parameters :

coalitionName	string	Coalition name : "NEUTRAL", "RED" ou "BLUE"
text	string	Text to display
duration	number	Duration in seconds.

Return values: None

ATME.displayForCountry(countryName, text, duration)

Description : This function displays a message to specific country players.

Parameters :

countryName	string	Country name
text	string	Text to display
duration	number	Duration in seconds.

Return values: None

Liste des nations (issue de DCS)

"RUSSIA", "UKRAINE", "USA", "TURKEY", "UK", "FRANCE", "GERMANY", "AGGRESSORS", "CANADA", "SPAIN", "THE_NETHERLANDS", "BELGIUM", "NORWAY", "DENMARK", "ISRAEL", "GEORGIA", "INSURGENTS", "ABKHAZIA", "SOUTH_OSETIA", "ITALY", "AUSTRALIA", "SWITZERLAND", "AUSTRIA", "BELARUS", "BULGARIA", "CHEZH_REPUBLIC", "CHINA", "CROATIA", "EGYPT", "FINLAND", "GREECE", "HUNGARY", "INDIA", "IRAN", "IRAQ", "JAPAN", "KAZAKHSTAN", "NORTH_KOREA", "PAKISTAN", "POLAND", "ROMANIA", "SAUDI_ARABIA", "SERBIA", "SLOVAKIA", "SOUTH_KOREA", "SWEDEN", "SYRIA"

ATME.explode(point, power)

Description : this function triggers an explosion at a specific point.

Parameters : None

point	table	x, y et z Point, or x,y Point
power	number	Explosion power

Return values: None

ATME.getAIUnits()

Description: Function retrieves all alive and active AI list at a T time.

Parameters : None

Return values:

	table	C_AIUnit active instance table
--	-------	--------------------------------

ATME.getFlag(id)

Description: function to retrieve a flag value. Value can be set by the mission editor.

Parameters :

id	number	Flag number
-----------	--------	-------------

Return values:

Without error :	number	Flag value
Si id incorrect :	nil	

ATME.getGroups()

Description : function to retrieve all alive and active group at T time

Parameters : None

Return values:

table	C_Group active instances table
-------	--------------------------------

ATME.getLanguage()

Description : function to retrieve the active language used for display.

Parameters : None

Return values:

string	Used language : "FR" french "EN" english
--------	--

ATME.getPlayers()

Description : function to retrieve all alive and active players at a T time.

Parameters : None

Return values:

table	C_Player alive instance table
-------	-------------------------------

ATME.getPointAltitude(point)

Description : function return altitude at a given point. (For a 2d Point x,y, return ground level.)

Parameters :

point	table	3D Point x, y et z, (or 2D Point x,y)
--------------	-------	---------------------------------------

Return values:

Without error :	number	Altitude in meter above sea level
Si point incohérent :	nil	

ATME.getSurfaceAltitude(point)

Description : Fonction retournant l'altitude du terrain à un point donné.

Parameters :

point	table	Point 3D x, y et z, ou 2D Point x,y
--------------	-------	-------------------------------------

Return values:

Without error :	number	Altitude in meter above sea level
Si point incohérent :	nil	

ATME.getTheatre()

Description : return activ map theatre mission.

Parameters : None

Return values:

string	Activ map name : "CAUCASUS" "NEVADA"
--------	--

ATME.getTime()

Description : Function to retrieve the number of seconds since mission start.

Parameters : None

Return values:

number	Number of seconds
--------	-------------------

ATME.getWindAzimuth(point3D, withTurbulence)

Description : Function return the horizontal direction of the wind at a given point.

Parameters :

Point3D	table	3D Point x, y et z
withTurbulence	boolean	If true takes account of turbulence, otherwise false.

Return values:

Without error :	number	Wind Azimuth (0 to 360°)
If error	nil	

ATME.getWindHSpeed(point3D, withTurbulence)

Description : Function return the horizontal speed of the wind at a given point.

Parameters :

Point3D	table	Point 3D x, y et z
withTurbulence	boolean	If true takes account of turbulence, otherwise false.

Return values:

Without error :	number	Wind speed m/s.
If error	nil	

ATME.getWindSpeed(point3D, withTurbulence)

Description : Function return speed of the wind at a given point. Return result on 3 axes

Parameters :

Point3D	table	3D Point x, y et z
withTurbulence	boolean	If true takes account of turbulence, otherwise false.

Return values:

Without error :	number	Wind speed m/s.
If error	nil	

ATME.getWindVector(point3D, withTurbulence)

Description : Function return speed vector of the wind at a given point.

Parameters :

Point3D	table	Point 3D x, y et z
withTurbulence	boolean	If true takes account of turbulence, otherwise false.

Return values:

Without error :	table	C_Vecteur3D Instance représentant le vecteur vitesse du vent m/s.
If error	nil	

ATME.getDCSZone(zoneName)

Description: Function to retrieve the center and radius information of a DCS zone defined in the mission.

Parameters :

zoneName	string	Zone name
-----------------	--------	-----------

Return values:

Without error :	table	zone table with center (zone.point) and radius (zone.radius) from the given zone
If error	nil	

ATME.isObjectClass(object, class)

Description : function to test if an object belong to a given class.

Parameters :

object	table	Objet
class	table	Class to test

Return values:

	boolean	True if object belong from given class , else false
--	---------	---

ATME.isPoint(tableToVerify)

Description : function to test if table point is 2D or 3D.

Parameters :

tableToVerify	table	Point table to test
----------------------	-------	---------------------

Return values:

number	ATME.POINT_2D for a 2D Point (x,y) ATME.POINT_3D for a 3D Point (x,y et z) ATME.POINT_BAD if table is not a point
--------	--

ATME.isStaticObject(objectName)

Description : Function to test if object given by its name is a static object.

Parameters :

objectName	string	Unit Name
-------------------	--------	-----------

Return values:

boolean	True if unit is AI, else false
---------	--------------------------------

ATME.isSurfaceLand(point)

Description : function test the ground type soil on surface (not suitable for testing runway see specific function)

Parameters :

point	table	3D x, y et z Point, or 2D Point x,y
--------------	-------	-------------------------------------

Return values:

Without error :	boolean	True if soil, else false
If error	nil	

ATME.isSurfaceRoad(point)

Description : Return if a point of the terrain is on a road.

Parameters :

point	table	3D Point x, y et z, or 2D Point x,y
--------------	-------	-------------------------------------

Return values:

Without error :	boolean	True if road , else false
If error	nil	

ATME.isSurfaceRunway(point)

Description : Return if a point of the terrain is on a runway.

Parameters :

point	table	3D Point x, y et z, or 2D Point x,y
--------------	-------	-------------------------------------

Return values:

Without error :	boolean	True if runway , else false
If error	nil	

ATME.isSurfaceWater(point)

Description : Return if a point of the terrain is on water

Parameters :

point	table	3D Point x, y et z, or 2D Point x,y
--------------	-------	-------------------------------------

Return values:

Without error :	boolean	True if water , else false
If error	nil	

ATME.isUnitControllable(unitName)

Description : Function to determine whether a unit given by its name is a player unit (active or not) as defined in the mission.

Parameters :

unitName	string	Unit Name
-----------------	--------	-----------

Return values:

	boolean	True if unit is controllable, else false
--	---------	--

ATME.loopTransmission(file, point, modulation, frequency, power)

Description : triggers a sound loop file at the selected radio frequency. The transmission source is located at the point in paramaters. The file is replayed systematically as soon as it ends. To stop the transmission, use the **ATME.stopTransmission** function. The sound file must be inside the mission miz or loaded in Mission editor once.

Parameters :

file	string	Sound file mame
point	table	3D Point x, y et z, or 2D Point x,y
modulation	string	"AM" or "FM"
frequency	number	transmission frequency Hz
power	number	Transmission power (W)

Return values:

Without error :	number	radio transmission Ident variable (used to stop it)
If error	nil	

Note : transmission power from 5 to 25 W

ATME.rotationH(point, center, angle)

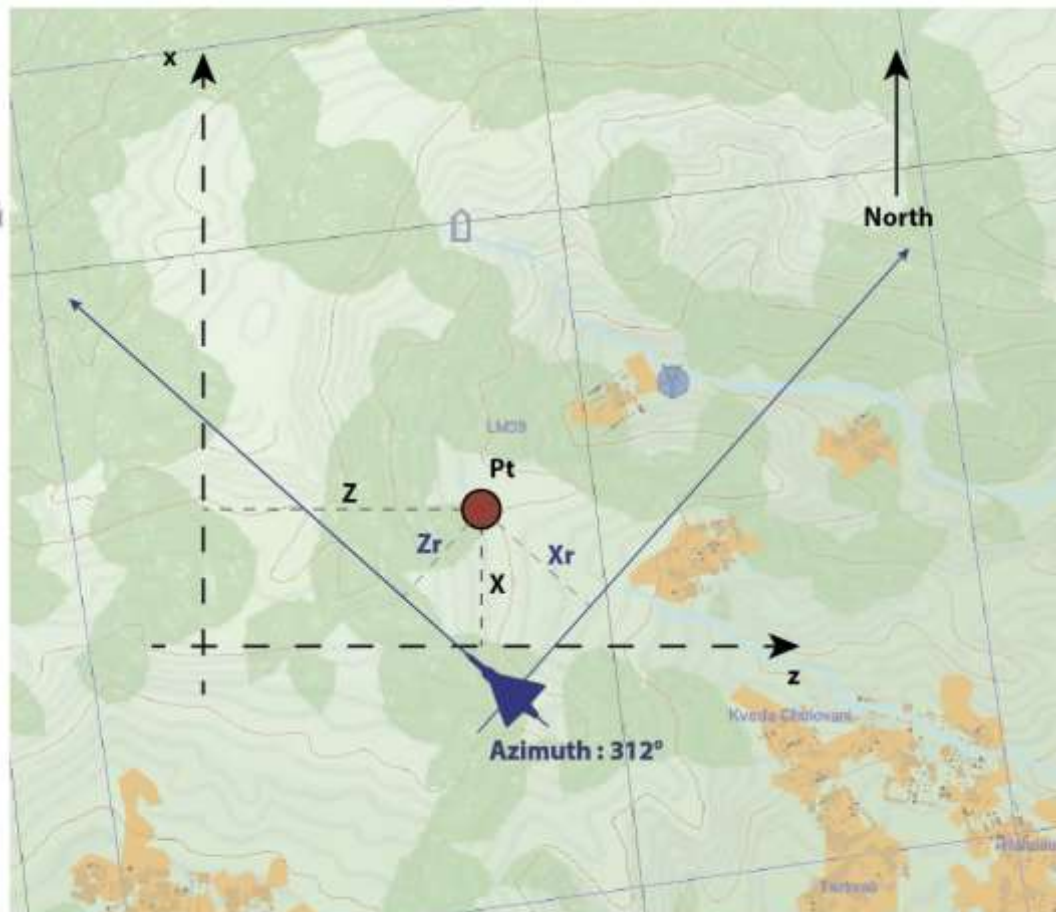
Description : this function convert a point in reference (center/angle) into a 3D Point in DCS reference frame (x,y,z).

Parameters :

point	table	3D Point x, y et z, or 2D Point x,y
center	table	3D Point x, y et z, or 2D Point x,y
angle	number	Rotation angle in degrees

Return values:

Without error :	table	3D Point x, y et z
If error	nil	



The aircraft has an azimuth of 312 ° (equivalent to -48 °, the axis X representing the north (Azimuth 0 °), and the coordinates of Pt in the reference frame of the aircraft are Zr and Xr, The function will allow to calculate the coordinates of Pt (Z, X) in the DCS frame. The plane will be the center of rotation and the angle of rotation will be -azimuth (48 ° for this case).

ATME.run(language)

Description : This function launch the ATME execution after loading all the necessary modules. It also sets the language. This function is set directly into a doscript() in the mission editor. also defines labels depending on the language chosen. If labels doesn't exist, an execution error may occur. The module Multilanguage definition requires a particular approach described in top manual. After its execution, it is no longer possible to load new modules.

Parameters :

language	string	Display language : "FR" : french "EN" : english "GER" : german "RUS" : russian
-----------------	--------	--

Return values: None

ATME.scalarHFromPoints(pA, pB, pC)

Description : This function calculates the scalar product of the AB and AC vectors 2D (x, z in DCS format).

Parameters :

pA	table	3D Point x, y et z, or 2D Point x,y
pB	table	3D Point x, y et z, or 2D Point x,y
pC	table	3D Point x, y et z, or 2D Point x,y

Return values:

Without error :	number	scalar product
If error	nil	

ATME.sendTransmissionOnce(file, point, modulation, frequency, power)

Description : triggers a sound file once at the selected radio frequency. The transmission source is located at the point in paramaters. The sound file must be inside the mission miz or loaded in Mission editor once

Parameters :

file	string	"Sound file Name"
point	table	3D Point x, y et z, or 2D Point x,y
modulation	string	"AM" or "FM"
frequency	number	transmission frequency Hz
power	number	Transmission power (W)

Return values: None

Note : transmission power from 5 to 25 W

ATME.setDebugLevel(level)

Description : This function is used to set the maximum level of debug messages display issued by the **C_ModuleInfos.output** method. Messages from higher levels will not be displayed. All messages are stored in **dcs.log** (inside savedgame/DCS), regardless of their level. This function is set directly into a doscript() in the mission editor during the tests. By default, the level is set to 0, which prevents any debug level display even if the module is in debug mode.

Parameters :

level	number	Maximum level of debug display
--------------	--------	--------------------------------

Return values: None

ATME.setF10groupIDAlphaOrder()

Description : modifies the ranking of player menu items. the items in an F10 menu are classified first by their group number and if the numbers are identical in alphabetical order.

Parameters : None

Return values: None

ATME.setF10AlphaOrder()

Description : modifies the ranking of player menu items. After executing this function, the items in an F10 menu are classified in alphabetical order. It is default ATME ranking.

Parameters : None

Return values: None

ATME.setFlag(id, value)

Description : Set a value to a flag

Parameters :

id	number	Flag number
value	boolean ou number	if boolean set 1 ou 0, Value is set as integer if nessary.

Return values: Nonee

ATME.soundForAll(file)

Description : Play a sound file to all players. The sound file must be inside the miz file or launch once in Mission editor.

Parameters :

file	string	Sound file name
-------------	--------	-----------------

Return values: None

ATME.soundForCoalition(coalition, file)

Description : plays a sound file to specific coalition players

Parameters :

coalitionName	string	Coalition name : "NEUTRAL", "RED" ou "BLUE"
file	string	Sound file name

Return values: None

ATME.soundForCountry(countryName, file)

Description : plays a sound file to specific country players

Parameters :

countryName	string	Country name
file	string	Sound file name

Return values: None

Country name (from DCS)

"RUSSIA", "UKRAINE", "USA", "TURKEY", "UK", "FRANCE", "GERMANY", "AGGRESSORS", "CANADA", "SPAIN", "THE_NETHERLANDS", "BELGIUM", "NORWAY", "DENMARK", "ISRAEL", "GEORGIA", "INSURGENTS", "ABKHAZIA", "SOUTH_OSETIA", "ITALY", "AUSTRALIA", "SWITZERLAND", "AUSTRIA", "BELARUS", "BULGARIA", "CHEZH_REPUBLIC", "CHINA", "CROATIA", "EGYPT", "FINLAND", "GREECE", "HUNGARY", "INDIA", "IRAN", "IRAQ", "JAPAN", "KAZAKHSTAN", "NORTH_KOREA", "PAKISTAN", "POLAND", "ROMANIA", "SAUDI_ARABIA", "SERBIA", "SLOVAKIA", "SOUTH_KOREA", "SWEDEN", "SYRIA"

ATME.startTransmission(file, point, modulation, frequency, power, interval)

Description : This function triggers a sound file transmission at regular intervals on selected radio frequency. The source of the transmission is located on point in parameter. If the time interval is less than the playing time, playback will be stopped to resume at the beginning. If the time interval is longer than the playing time, no sound will be output after playback until next trigger. To stop the transmission, use the **ATME.stopTransmission** function. The sound file must be inside the mission miz, or launched once in Mission editor

Parameters :

file	string	Sound file name
point	table	3D Point x, y et z, or 2D Point x,y
modulation	string	"AM" or "FM"
frequency	number	Transmission frequency Hz
power	number	Transmission power (W)
interval	number	Interval in seconds to emit. 0 value will do once. interval is set as integer if needed.

Return values:

Without error :	number	radio transmission ident variable
If error	nil	

Note : 5W or 25W power

ATME.stopTransmission(id)

Description : this function stop a radio transmission in loop or once.

Parameters :

id	number	radio transmission ident variable grab from
		startTransmission / loopTransmission on
		return value

Return values: None

ATME.whichSide(pA, pB, pC)

Description : return the relative position (right or left) of the point C relative to the oriented line AB.

Parameters :

pA	table	3D Point x, y et z, or 2D Point x,y
pB	table	3D Point x, y et z, or 2D Point x,y
pC	table	3D Point x, y et z, or 2D Point x,y

Return values:

Without error :	number	-1 : C is left from AB 1 : C is right from AB 0 : C is on AB
If error	nil	

Fonctions de création d'objets statiques de la table ATME

```
ATME.staticObjects.createCargo(countryName, name, type, position, mass, isDead)
```

Description : This function creates a static Cargo object dynamically (spawn). azimuth is determined randomly. The position will be a specific point or a random point in a DCS zone set in parameter field. It will have a given mass (mass) and will be either "alive" (isDead = false) or destroyed (isDead = true).

Parameters :

countryName	string	Country name
name	string	Object name
position	table	3D Point x, y et z, or 2D Point x,y. DCS zone Name if random spawn
type	string	Cargo type (see below)
mass	number	Mass in Kg
isDead	boolean	True or false

Return values: None

Country names (from DCS)

"RUSSIA", "UKRAINE", "USA", "TURKEY", "UK", "FRANCE", "GERMANY", "AGGRESSORS", "CANADA", "SPAIN", "THE_NETHERLANDS", "BELGIUM", "NORWAY", "DENMARK", "ISRAEL", "GEORGIA", "INSURGENTS", "ABKHAZIA", "SOUTH_OSETIA", "ITALY", "AUSTRALIA", "SWITZERLAND", "AUSTRIA", "BELARUS", "BULGARIA", "CHEZH_REPUBLIC", "CHINA", "CROATIA", "EGYPT", "FINLAND", "GREECE", "HUNGARY", "INDIA", "IRAN", "IRAQ", "JAPAN", "KAZAKHSTAN", "NORTH_KOREA", "PAKISTAN", "POLAND", "ROMANIA", "SAUDI_ARABIA", "SERBIA", "SLOVAKIA", "SOUTH_KOREA", "SWEDEN", "SYRIA"

Cargo types :

"BASIC" for the first original type (only 2.0.4 available type), "TRUNKS SMALL", "TRUNKS LONG", "TETRAPOD", "PIPE SMALL", "PIPE BIG", "OIL TANK", "M117", "SMALL CONTAINER", "F BAR", "FUEL TANK", "BW CONTAINER", "CONTAINER", "BARRELS", "AMMO BOX"

```
ATME.staticObjects.createWhiteContainer(countryName, name, position, heading, isDead)
```

Description : This function creates a static White container object dynamically (spawn). The position will be a specific point or a random point in a DCS zone set in parameter field. It will be either “alive” (isDead = false) or destroyed (isDead = true).

Parameters :

countryName	string	Country Name
name	string	Object Name
position	table	3D Point x, y et z, or 2D Point x,y. DCS zone Name if random spawn
heading	number	Object Azimuth in degrees
isDead	boolean	True or false

Return values: None

Country names :

"RUSSIA", "UKRAINE", "USA", "TURKEY", "UK", "FRANCE", "GERMANY", "AGGRESSORS", "CANADA", "SPAIN", "THE_NETHERLANDS", "BELGIUM", "NORWAY", "DENMARK", "ISRAEL", "GEORGIA", "INSURGENTS", "ABKHAZIA", "SOUTH_OSETIA", "ITALY", "AUSTRALIA", "SWITZERLAND", "AUSTRIA", "BELARUS", "BULGARIA", "CHEZH_REPUBLIC", "CHINA", "CROATIA", "EGYPT", "FINLAND", "GREECE", "HUNGARY", "INDIA", "IRAN", "IRAQ", "JAPAN", "KAZAKHSTAN", "NORTH_KOREA", "PAKISTAN", "POLAND", "ROMANIA", "SAUDI_ARABIA", "SERBIA", "SLOVAKIA", "SOUTH_KOREA", "SWEDEN", "SYRIA"

ATME Class

ATME objects have methods related to their class or instance. In the following, the name of the class followed by ". " Will be mentioned for **class methods** and the word Object followed by": " for **functions** related to the instances of this class.

ATME.C_AIUnit class

instances of this class are created and deleted by ATME Core.

ATME.C_AIUnit.exists(name)

Description : function to test if a AI unit exists.

Parameters :

name	string	AI unit Name
-------------	--------	--------------

Return values:

Without error :	boolean	True if C_AIUnit instance exists, else false
If error	nil	

ATME.C_AIUnit.getByname(name)

Description : retrieve a alive AI by its Name.

Parameters :

name	string	AI Unit Name
-------------	--------	--------------

Return values:

Without error :	table	C_AIUnit instance class table
If error	nil	

Object:crossAxisFromLeftToRight(pA, pB)

Description : This method test the crossing by the AI unit of a horizontal line (straight line 2D on the x, z axes) represented by the points A and B. The direction AB is oriented and makes it possible to define the crossing direction, here from left to right.

Parameters :

pA	table	3D Point x, y et z, or 2D Point x,y
pB	table	3D Point x, y et z, or 2D Point x,y

Return values:

Without error :	boolean	True if crossing in correct direction is done , else false
If error	nil	

Object:crossAxisFromRightToLeft(pA, pB)

Description : This method test the crossing by the AI unit of a horizontal line (straight line 2D on the x, z axes) represented by the points A and B. The direction AB is oriented and makes it possible to define the crossing direction, here from right to left.

Parameters :

pA	table	3D Point x, y et z, ou2D Point x,y
pB	table	3D Point x, y et z, ou2D Point x,y

Return values:

Without error :	boolean	True if crossing in correct direction is done , else false
If error	nil	

Object:disable()

Description : This method disables an AI unit. Allow to delete a unit from the game without triggering DCS event S_EVENT_DEAD.

module handlers activated :

- onDisableAIUnitHandler
- onDeleteAIUnitHandler with parameter isEnabled = false

Parameters : None

Return values: None

Object:explode(power)

Description : trigger an explosion on a Ai Unit. usage AiUnit:explode()

Parameters :

power	number	Explosion power
--------------	--------	-----------------

Return values: None

Object:fireFlare(color)

Description : This method triggers flare firing from AI unit.

Parameters :

color	string	flare colour "GREEN", "RED", "WHITE" et "YELLOW". "RANDOM" value
--------------	--------	---

Return values: None

Object:fireIlluminationBomb(power)

Description : This method triggers Illumination Bomb firing from AI unit.

Parameters :

power	number	Illumination bomb power
--------------	--------	-------------------------

Return values: None

Object:fireSmoke(color)

Description : This method triggers a smoke near a AI unit.

Parameters :

color	string	Smoke colour "BLUE" , "GREEN", "RED", "WHITE" et "ORANGE". "RANDOM"
--------------	--------	---

Return values: None

Object:getAGLAltitude()

Description : This method retrieve altitude above ground level from a AI Unit.

Parameters : None

Return values:

number	altitude in meters
--------	--------------------

Object:getAzimuth()

Description : This method retrieves Ai unit azimuth (relative to the north). This corresponds to the true heading without taking into account the magnetic deviation.

Parameters : None

Return values:

number	Azimuth in degrees (0-360)
--------	----------------------------

Object:getCallsign()

Description : This method retrieves the name used inside coalition for the AI unit. The callsign consists of « nom id1-id2 », exemple « enfield 1-1 »

Parameters : None

Return values:

string	Callsign name
string	callsign Id1
string	callsign Id2

Object:getClassName()

Description : This method retrieves the name of the object ATME class, in this case "C_AIUnit". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object: getCoalitionName()

Description : This method retrieves the coalition name from the AI unit.

Parameters : None

Return values:

Without error :	string	Coalition name : "RED", "BLUE" or "NEUTRAL"
-----------------	--------	---

Object: getCountryName()

Description : This method retrieves the country name from the AI unit.

Parameters : None

Return values:

string	Country name
--------	--------------

Countries names from DCS :

"RUSSIA", "UKRAINE", "USA", "TURKEY", "UK", "FRANCE", "GERMANY", "AGGRESSORS", "CANADA", "SPAIN", "THE_NETHERLANDS", "BELGIUM", "NORWAY", "DENMARK", "ISRAEL", "GEORGIA", "INSURGENTS", "ABKHAZIA", "SOUTH_OSETIA", "ITALY", "AUSTRALIA", "SWITZERLAND", "AUSTRIA", "BELARUS", "BULGARIA", "CHEZH_REPUBLIC", "CHINA", "CROATIA", "EGYPT", "FINLAND", "GREECE", "HUNGARY", "INDIA", "IRAN", "IRAQ", "JAPAN", "KAZAKHSTAN", "NORTH_KOREA", "PAKISTAN", "POLAND", "ROMANIA", "SAUDI_ARABIA", "SERBIA", "SLOVAKIA", "SOUTH_KOREA", "SWEDEN", "SYRIA"

Object:getDCSUnit()

Description : This method retrieves the AI unit as managed by DCS (Unit Class Object).

Parameters : None

Return values:

table	DCS Unit instance class table
-------	--------------------------------------

Object:getFuelRatio()

Description : This method allows to know the ratio of fuel remaining to full.

Parameters : None

Return values:

number	Fuel remaining ratio
--------	----------------------

Object:getGroup()

Description : This method retrieves the **ATME.C_Group** instance corresponding to the AI.

Parameters : None

Return values:

table	ATME.C_Group instance class table for the AI
-------	---

Object: getGroupsNameOnBoard()

Description : This method retrieves the list of infantry group names embedded in the AI unit. The AI unit must be an authorized vehicle, that is to say it has a capacity to transport troops (personnel carrier).

Parameters : None

Return values:

table	Indexed table with groups on board names. The first embedded group is at index 1.
-------	---

Object: getHSpeed()

Description : This method recovers horizontal speed of the AI Unit in m / s (calculated on 2 axes x and z).

Parameters : None

Return values:

number	AI horizontal speed m/s
--------	-------------------------

Object: getLife()

Description : This method retrieves the "life" information from an AI unit.

Parameters : None

Return values:

number	AI life information
--------	---------------------

Object: getLifeRatio()

Description : This method retrieves the "life" ratio from an AI unit.

Parameters : None

Return values:

number	Life ratio
--------	------------

Object: getMSLAltitude()

Description : This method recovers the AI Unit altitude relative to sea level.

Parameters : None

Return values:

number	Ai Unit Altitude in meters
--------	----------------------------

Object: getName()

Description : this method retrieves UnitAI Name

Parameters : None

Return values:

string	AI Unit Name
--------	--------------

Object: getNbUnitsOnBoard()

Description : This method recovers the number of infantry on board.

Parameters : None

Return values:

number	number of infantry on board
--------	-----------------------------

Object: getNearestReadyToBoardGroups(radius)

Description : This method search the nearest embarkable infantry group within the specified radius.
If no unit available, nil will be returned.

Parameters :

radius	number	Search radius
---------------	--------	---------------

Return values:

	table	nearest embarkable infantry group (ATME.C_Group)
	nil	If not available

Object:getPosition()

Description : This method retrieve AI Unit position (point x y z)

Parameters : None

Return values:

	table	3D Point x, y et z
--	-------	--------------------

Object:getRollAxisVector()

Description : This method returns a vector corresponding to the AI unit longitudinal axis, facing forward.

Parameters : None

Return values:

	table	ATME.C_Vector3D type vector
--	-------	------------------------------------

Object:getSpeed()

Description : This method recovers the AI speed in m / s (calculated on the 3 axes).

Parameters : None

Return values:

number	AI Unit speed m/s
--------	-------------------

Object:getSpeedAzimuth()

Description : This method recover AI unit true route (heading of the velocity vector on the x and z axes)

Parameters : None

Return values:

number	Heading true route in degrees (0-360)
--------	---------------------------------------

Object:getTypeName()

Description: This method retrieves AI unit type name.

Parameters: None

Return values:

string	AI Unit type Name
--------	-------------------

Object:getVelocityVector()

Description : This method recovers AI unit speed vector

Parameters : None

Return values:

table	C_Vector3D instance for speed
-------	-------------------------------

Object:getVSpeed()

Description: This method recovers AI unit vertical speed (y axes). m/s

Parameters : None

Return values:

number	AI Unit vertical speed m/s
--------	----------------------------

Object:inAir()

Description : this method test if Ai Unit is in Air.

Parameters : None

Return values:

boolean	True or false
---------	---------------

Object:isAAA()

Description : This method is used to determine whether the AI unit is an anti-aircraft gun unit.

Parameters : None

Return values:

boolean	True or false
---------	---------------

Object:isAirDefence()

Description : This method is used to determine whether the AI unit is an air defense unit.

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isGroundVehicle()

Description : This method is used to determine whether the AI unit is a ground vehicle.

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isHelicopter()

Description : This method is used to determine whether the AI unit is a helicopter.

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isInfantry()

Description : This method is used to determine whether the AI unit is an infantry unit.

Parameters : None

Return values:

	boolean	True / false
--	---------	--------------

Object:isInDCSZone(zoneName)

Description : This method test if AI unit is in a DCS zone defined in the mission editor.

Parameters : None

zoneName	string	DCS Zone Name
-----------------	--------	---------------

Return values:

Without error :	boolean	True / false
If error	nil	

Object:isInZone2D(reference, radius)

Description: This method determines whether the AI unit is in a horizontal zone defined by a Unit a circle (center radius). The center can be :

- 2D Point or 3D point,
- A AI Unit position
- A player Unit position
- A static Object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit. Type ATME.C_Player Static Object Type ATME.C_StaticObject
radius	number	Circle radius in meter

Return values:

Without error :	boolean	True if AIUnit is in horizontal zone else false
If error	nil	

Object:isInZone3D(reference, radius)

Description : This method determines whether the AI unit is in a spherical zone defined by a Unit a sphere (center radius). The center can be :

- 2D Point or 3D point,
- A AI Unit position
- A player Unit position
- A static Object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject
radius	number	Spherical radius in meter

Return values:

Without error :	boolean	True if AIUnit is in spherical zone else false
If error	nil	

Object:isNear(reference, radius, deltaAltitude)

Description : This method test if a reference is close to the AI unit (object). The zone is defined by a cylinder represented by a horizontal circle, the two sides of the cylinder correspond to an altitude difference (the unit AI as the center).

Center can be :

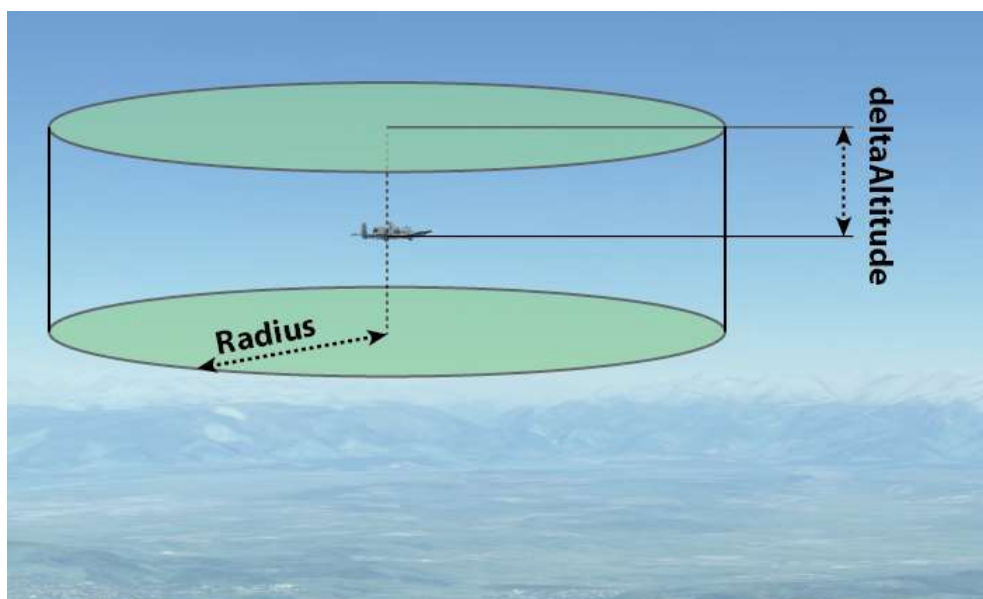
- 2D Point or 3D point,
- A AI Unit position
- A player Unit position
- A static Object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static object Type ATME.C_StaticObject
radius	number	Circle radius top and bottom in meters
deltaAltitude	number	Altitude difference between top/bottom and center m

Return values:

Without error :	boolean	If true, reference is in cylinder else false.
If error	nil	



Object:isManPad()

Description : This method is used to determine whether the AI unit is an infantry equipped with anti-aircraft missiles.

Parameters : None

Return values:

boolean True / false

Object:isPersonnelCarrier()

Description : This method test if the AI unit is able to transport troops.

Parameters : None

Return values:

boolean True if personnal carrier else false

Object:isPlane()

Description: This method test if the AI unit is an aircraft.

Parameters : None

Return values:

boolean True / false

Object:isSAMVehicle()

Description : This method test if AIUnit Is a ground to air missile unit

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isSAMSiteCommandCenter()

Description : This method test if AIUnit Is a command unit for ground to air missile site.

Parameters : None

Return values:

boolean	True /false
---------	-------------

Object:isSAMSiteLauncher()

Description : This method test if AIUnit is a SAM launcher.

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isSAMSiteRadar()

Description : This method test if AIUnit is a radar Unit of a Ground/air missile installation

Parameters : None

Return values:

boolean True / false

Object:load(groupToBoard, radius)

Description : This method allow to embark an infantry group. AI unit must be an authorized vehicle, : has a capacity to transport troops (personnel carrier). The group to be embarked must be within the boarding radius of the AI unit (radius for embarkation). The troop transport AI unit must also be on place, and have a horizontal speed of less than 1m / s.

A troop transport vehicle has a maximum boarding capacity of infantry units. It can carry several infantry groups as long as this maximum capacity is not exceeded. The total mass of transport is currently unchanged.

The ATME Core event "TRANSPORT_END_OF_BOARDING" is complete at the end of embarkation.

Parameters :

groupToBoard	table	Infantry group to board (ATME.C_Group)
radius	number	Radius limit to board

Return values:

Without error :	boolean	False if no error
If error	boolean	True if error
	string	Error codes :
		"LOAD_UNIT_TOO_FAR"
		"LOAD_UNIT_IN_AIR"
		"LOAD_UNIT_BAD_SPEED"
		"LOAD_TOO_MUCH_INFANTRY"
		"LOAD_INFANTRY_GROUP_NOT_READY"
		"LOAD_NO_INFANTRY_GROUP_AVAILABLE"

Allowed to transport troops helicopters :

"UH-1H" 8 infantry units limit, "Mi-8MT" 16 infantry units limit, "SA342" 2 infantry unit limit,

Allowed to transport troops ground vehicules :

"AAV7" 25 infantry units limit, "M-113" 11 infantry units limit, "LAV-25" 6 infantry units limit, "M1126 Stryker ICV" 9 infantry units limit, "M-2 Bradley" 6 infantry units limit, "BTR-80" 9 infantry units limit, "BTR_D" 12 infantry units limit, "MTLB" 10 infantry units limit, "BMD-1" 4 infantry units limit, "BMP-1" 8 infantry units limit, "BMP-2" 7 infantry units limit, "BMP-3" 7 infantry units limit, "UAZ-469" 6 infantry units limit, "Tigr_233036" 6 infantry units limit, "GAZ-3307" 16 infantry units limit, "GAZ-3308" 16 infantry units limit, "Ural-4320T" 20 infantry units limit, "Ural-4320-31" 20 infantry units limit, "M 818" 20 infantry units limit, "KAMAZ Truck" 20 infantry units limit

Object:loopTransmission(file, modulation, frequency, power)

Description : This method triggers a sound file loop on the selected radio frequency. The file is replayed as soon as its ends. To stop the transmission, use the **Object: stopTransmission** function. The sound file must be defined in the mission. The transmission source corresponds to the unit position when the function is called. If a unit triggers several calls to this function, the last call replaces the previous calls.

Parameters :

file	string	Sound file Name
modulation	string	"AM" or "FM"
frequency	number	Transmission frequency Hz
power	number	Transmission power (W)

Return values:

Without error :	number	Ident variable used in Object:stopTransmission
If error	nil	

Note : transmission power 5 to 25W

Object:sendTransmissionOnce(file, modulation, frequency, power)

Description : This method triggers a single sound file transmission on the selected radio frequency. The transmission source corresponds to the Unit position when the function is called

Parameters :

file	string	Sound file Name
modulation	string	"AM" or "FM"
frequency	number	Transmission frequency Hz
power	number	Transmission power (W)

Return values: None

Object:startTransmission(file, modulation, frequency, power, interval)

Description : This method triggers a sound file transmission at regular intervals on the selected radio frequency. If the time interval is less than the playing time, playback will be stopped to resume at the beginning. If the time interval is longer than the playing time, no sound will be output after playback until the re-trigger. To stop the transmission, use the **Object: stopTransmission** function. The sound file must be defined in the mission. The transmission source corresponds to the Unit position when the function is called. If a unit triggers several calls to this function, the last call replaces the previous ones.

Parameters :

file	string	Sound file Name
modulation	string	"AM" or "FM"
frequency	number	Transmission frequency Hz
power	number	Transmission power (W)

Return values:

Without error :	number	Ident variable used in Object:stopTransmission
If error	nil	

Note : power from 5 to 25 W

Object:stopAllTransmissions()

Description : This method stops all transmission loop or transmission at regular intervals on a given Object

Parameters : None

Return values: None

Object:stopTransmission(id)

Description : This method stops specific transmission loop or transmission at regular intervals on a given Object. Id of transmission must be specified, it is returned by :

- **Object:startTransmission**
- **Object:loopTransmission**

Parameters :

id	number	Specific Ident Variable from the transmission
-----------	--------	---

Return values: None

Object:unload(id)

Description : This method is used to disembark an infantry group present in a troops transport Units. Id corresponds to the index in the order of embarkation, if necessary, it will be necessary to use **getGroupsNameOnBoard** in order to retrieve the indexed list of the troops on board. Id must match one of the indexes in that list of names.

ATME Core event "TRANSPORT_END_OF_DISEMBARK" is launch at end of boarding

Parameters :

id	number	Boarding Index
-----------	--------	----------------

Return values:

Without error :	boolean	False if no error
	table	Group instance unloaded
	number	Lost unit in unload
If error	boolean	True if errors
	string	Errors code :
		"UNLOAD_UNIT_IN_AIR"
		"UNLOAD_UNIT_BAD_SPEED"
		"UNLOAD_BAD_ID"

Object:whichSide(reference)

Description : This function retrieves the relative position (right or left) from the reference with respect to the unit AI :

- 2D Point or 3D point,
- unit AI position
- Player Unit position
- Static Object position

Parameters :

reference	table	3D Point x, y et z, or 2D Point x,y Unit AI Type ATME.C_AIUnit Player Unit. Type ATME.C_Player Static Objet Type ATME.C_StaticObject
------------------	-------	--

Return values:

Without error :	number	-1 : reference is on the left from AIUnit 1 : reference is on theright from AIUnit 0 : reference is on Axis from AiUnit
If error	nil	

Classe ATME.C_AirBase

Instances of this class are created and deleted by ATME Core.

All bases are referenced whether they are linked to the map or created in the mission (FARP or Airfield)

Object:getName()

Description : This method allows to retrieve the name of the object ATME class, in this case "C_Airbase". This method exists for all classes

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getName()

Description : This method retrieves the name of a base.

Parameters : None

Return values:

string	base Name
--------	-----------

ATME.C_EventMgr Class

The **C_EventMgr** class is the ATME event handling class. Instances of this class are handled directly by ATME and send parameter (events) to module handlers (see class **C_Module** for more information):

- **onUpdatePlayerHandler(player, events)**
- **onTimerHandler(events)**

A user module can not create an instance of this class.

There are two types of events managed by this class:

- **Core events:** They are generated by ATME Core and sent to all modules active in the mission. These events concern entry / exit of surveillance zones, embarkation end / disembarkation and race events.
- Events associated with **user triggers**: These events are generated by ATME based on the individual state of the user triggers and sent to the module that created this user trigger. Other modules will not receive the event.

An event is sent only once to a module. If it is not processed by the module, it will be lost. Only events from user triggers with a long duration of activation can be generated and sent several times until the end of activation.

The identifier (**id**) of an event is:

- A number if the event is a Core event
- A string if the event is generated by a user trigger. It is the associated user trigger name

Object: getCoreEventType(id)

Description : This method retrieves the Core event type. Id is given by a loop using the associated peer function.

Each Core event has associated data (**datas**) that varies according to the type of the event.

Parameters :

id	number	Event Id
-----------	--------	----------

Return values:

string	Event core type
--------	-----------------

Types of Core events associated with races (see also **C_Race** for more information):

- **"RACE_PLAYER_TOO_HIGH"** : Generated if the player is too high passing through a door
- **"RACE_FINAL_TIME"** : Generated when a player crosses the finish gate. The race is over for this player.
- **"RACE_BEST_FINAL_TIME"** : Generated when a player achieves the best race time
- **"RACE_PLAYER_BEST_FINAL_TIME"** : Generated when a player achieves his best time
- **"RACE_BEST_LAP_TIME"** : Generated when a player achieves the best intermediate time running at a given gate.
- **"RACE_LAP_TIME"** : Generated when a player crosses a gate with an intermediate time measure
- **"RACE_BEST_TURN_TIME"** : Generated when a player achieves the best lap race. This is only valid for multi-lap races.
- **"RACE_PLAYER_BEST_TURN_TIME"** : Generated when a player achieves his best lap race. This is only valid for multi-lap races.
- **"RACE_TURN_TIME"** : Generated when a player crosses the starting gate after one full turn. This is only valid for multi-lap races.
- **"RACE_DOOR_MISSED"** : Generated when a player misses a door.
- **"RACE_START"** : Generated when a player crosses the gate and starts his race. For races with several rounds, this event will be generated only once, beginning of race

Event Types Core Associated with entry/exit in zone :

- **"SIGNAL_UNIT_IN_ZONE"** : Generated when a troop transport unit enters a zone.
- **"SIGNAL_UNIT_OUT_OF_ZONE"** : Generated when a troop transport unit exits a zone

Core event types associated with troop transport:

- **"TRANSPORT_END_OF_BOARDING"** : Generated when a troop transport unit has finished embarking an infantry group.
- **"TRANSPORT_END_OF_DISEMBARK"** : Generated when a troop transport unit has finished disembarking an infantry group.

Object: `getCoreEventDatas(id)`

Description : This method retrieves datas from a Core event. Each Core event has associated data (datas) that varies according to the type of the event.

Parameters :

id	number	Event Id
-----------	--------	----------

Return values:

table	Associated datas from Event
-------	-----------------------------

Below, the return value is associated with the variable **datas** :

```
local datas = Object:getCoreEventDatas(id)
```

Types of Core events associated with races (see also **C_Race** for more information):

- "RACE_PLAYER_TOO_HIGH" :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.penalty** : stock penalty for player who initiate the event.
- "RACE_FINAL_TIME" :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.penalty** : stock penalty for player who initiate the event.
 - **datas.totalPlayTime** : Total time for the player
- "RACE_BEST_FINAL_TIME" :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator

- **datas.totalPlayTime** : total time become best final time in race
- "RACE_PLAYER_BEST_FINAL_TIME" :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.totalPlayTime** : total time became the final player time
- "RACE_BEST_LAP_TIME" :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.playTime** : lap time
 - **datas.turnNumber** : lap Number
 - **datas.doorNumber** : door number set on time check
- "RACE_LAP_TIME" :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.penalty** : stock penalty for the player initiator
 - **datas.playTime** : lap time
 - **datas.totalPlayTime** : total time since race start
 - **datas.turnNumber** : Lap Number.
 - **datas.doorNumber** : door number set on time check
- "RACE_BEST_TURN_TIME" :
 - **datas.race** : race event initiator

- **datas.unit** : Player (**C_Player** instance) event initiator
- **datas.playTime** : lap time become best lap time in race
- **datas.turnNumber** : lap number
- **"RACE_PLAYER_BEST_TURN_TIME"** :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.playTime** : lap time become best lap time for player
 - **datas.turnNumber** : lap number
- **"RACE_TURN_TIME"** :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.penalty** : stock penalty for the player.
 - **datas.playTime** : time present lap
 - **datas.totalPlayTime** : total time from race start
 - **datas.turnNumber** : lap number
- **"RACE_DOOR_MISSED"** :
 - **datas.race** : race event initiator
 - **datas.unit** : Player (**C_Player** instance) event initiator
 - **datas.doorNumber** : door Number which was missed
- **"RACE_START"** :
 - **datas.race** : race event initiator

- **datas.unit** : Player (**C_Player** instance) event initiator

Event Types Core Associated with entry/exit in zone :

- **"SIGNAL_UNIT_IN_ZONE" :**
 - **datas.unit** : (**C_Player** ou **C_AIUnit** Instance) Player or Unit event initiator
 - **datas.linkedGroup** : (Instance **C_Group**) group checked
- **"SIGNAL_UNIT_OUT_OF_ZONE" :**
 - **datas.unit** : (**C_Player** ou **C_AIUnit** Instance) Unit event initiator
 - **datas.linkedGroup** : (Instance **C_Group**) group checked

Core event types associated with troop transport:

- **"TRANSPORT_END_OF_BOARDING" :**
 - **datas.unit** : (**C_Player** ou **C_AIUnit** Instance) Player or Unit event initiator
 - **datas.onBoardGroupName** : group name boarded
 - **datas.nbUnitsOnBoard** : Total of Unit on board
 - **datas.nbDeadUnits** : losed unit on embark
- **"TRANSPORT_END_OF_DISEMBARK" :**
 - **datas.unit** : (**C_Player** ou **C_AIUnit** Instance) Player or Unit event initiator
 - **datas.index** : Index of group just disembarked
 - **datas.linkedGroup** : (Instance **C_Group**) group just disembarked
 - **datas.nbUnitsOnBoard** : number of infantry unit still on board
 - **datas.nbDeadUnits** : losed unit on disembark

Object:isCoreEvent(id)

Description : This method test if an event is a Core event. Id is given by a loop using the associated peer function.

Parameters :

id	number	Event id
-----------	--------	----------

Return values:

	boolean	True if event is CORE elst false
--	---------	----------------------------------

Object:isUserTriggerEvent(id)

Description : This method test if an event is a User Trigger event. Id is given by a loop using the associated peer function.

Parameters :

id	string	Event Id (user trigger)
-----------	--------	-------------------------

Return values:

	boolean	True if User Trigger event else False
--	---------	---------------------------------------

Object:isUserTriggerEventToggle(id)

Description : This method test if an event state generated by a user trigger has just changed (the associated trigger has just been activated). Id is given by a loop using the associated peer function and matches the name of the user trigger.

This function is useful if a user trigger has a long duration. In this case, **isUserTriggerEventToggle** will return true only at the first handler call following its activation, unlike **isUserTriggerEvent**.Parameters :

id	string	Event Id (user Trigger)
-----------	--------	---------------------------

Return values:

	boolean	True if event from user trigger has just been activated else false.
--	---------	---

Object: pairs()

Description : This method redefines the method for scanning entries in a table. It use a loop to successively process all events received at a T time.

Parameters : None

Return values: linked with pairs (lua)

ATME.C_Flare class

ATME.C_Flare(colorName, point)

Description : launch a flare at a given point

Parameters :

colorName	string	Flare colour: "GREEN", "RED", "WHITE" et "YELLOW". "RANDOM"
point	table	3D Point x, y et z.

Return values:

Without error	table	C_Flare new instance table
Si erreur	nil	

Object: fire(duration)

Description : This method triggers random flares during a given period of time.

Parameters :

duration	number	Duration seconds
-----------------	--------	------------------

Return values: None

Object: fireOnce()

Description : lauch once a flare

Parameters : None

Return values: None

Object:getName()

Description : retrieves the Object name of the ATME class, in this case "C_Flare". This method exists for all classes

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getColor()

Description : retrieve flare colour

Parameters : None

Return values:

string	Flare colour string : "GREEN", "RED", "WHITE" ou "YELLOW".
--------	--

Object:getDuration()

Description : method to retrieve duration of flare fire.

Parameters : None

Return values:

number	Duration seconds
--------	------------------

Object:getPoint()

Description : method to retrieve the point where the flare was fired.

Parameters : None

Return values:

table	3D Point x, y et z, or 2D Point x,y
-------	-------------------------------------

Object:getTimeStart()

Description : retrieves time of fire start from mission start

Parameters : None

Return values:

number	Number of seconds since mission start
--------	---------------------------------------

Object:stop()

Description : This method stop a fire (**fire** function).

Parameters : None

Return values: None

ATME. C_Group Class

instances of this class are created and deleted by ATME Core.

ATME.C_Group.exists(name)

Description : Function checking whether a group is alive at a T instant (at least one life unit in the group)

Parameters :

name	string	Groupe Name
-------------	--------	-------------

Return values:

Without error :	boolean	True if C_Group instance exist, else false
If error	nil	

ATME.C_Group.getByName(name)

Description : Function to retrieve, from its name, a group alive at a T time.

Parameters :

name	string	Group name
-------------	--------	------------

Return values:

Without error :	table	C_Group class instance
If error	nil	

ATME.C_Group.getGroupsInDCSZoneForAll(zoneName, allGroup)

Description : This method returns all groups in a DCS zone defined in the mission editor. It is possible to define whether only the groups entirely and / or partially in the zone are taken into account.

Parameters :

zoneName	string	DCS zone Name
allGroup	boolean	If true, groups must be completely in the zone. If false, it one unit of the groups must be in zone.

Return values:

Without error :	table	(ATME.C_Group) group table. Index are group names. Table will be nil if no groups are inside zone.
If error	nil	

ATME.C_Group.getGroupsInDCSZoneForCoalition(coalitionName, zoneName, allGroup)

Description : This method returns all groups from a specific coalition, in a DCS zone defined in the mission editor. It is possible to define whether only the groups entirely and / or partially in the zone are taken into account.

Parameters :

coalitionName	string	coalition name : "RED", "BLUE" ou "NEUTRAL"
zoneName	string	DCS Zone Name
allGroup	boolean	If true, groups must be completely in the zone. If false, it one unit of the groups must be in zone.

Return values:

Without error :	table	(ATME.C_Group) group table. Index are group names. Table will be nil if no groups are inside zone.
If error	nil	

ATME.C_Group.getGroupsInZone2DForAll(reference, radius, allGroup)

Description : This method returns all groups, in a 2D zone. It is possible to define whether only the groups entirely and / or partially in the zone are taken into account.

Center of zone can be :

- 2D Point or 3D point
- AI unit position
- A player unit position
- A static object position

Parameters :

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static object Type ATME.C_StaticObject
radius	number	Radius zone 2D
allGroup	boolean	If true, groups must be completely in the zone. If false, one unit of the groups must be in zone.

Return values:

Without error :	table	(ATME.C_Group) group table. Index are group names. Table will be nil if no groups are inside zone.
If error	nil	

ATME.C_Group.getGroupsInZone2DForCoalition(coalitionName, reference, allGroup)

Description : This method returns all groups, in a 2D zone from a specific coalition. It is possible to define whether only the groups entirely and / or partially in the zone are taken into account.

Center of zone can be :

- 2D Point or 3D point
- AI unit position
- A player unit position
- A static object position

Parameters :

coalitionName	string	Coalition Name : "RED", "BLUE" ou "NEUTRAL"
reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static object Type ATME.C_StaticObject
radius	number	Radius zone 2D
allGroup	boolean	If true, groups must be completely in the zone. If false, one unit of the groups must be in zone.

Return values:

Without error :	table	(ATME.C_Group) group table. Index are group names. Table will be nil if no groups are inside zone.
If error	nil	

ATME.C_Group.getReadyToBoardGroupsForAll()

Description : this method retrieves all groups ready to board.

Parameters : None

Return values:

Without error :	table	(ATME.C_Group) group table. Index are group names. Table will be nil if no groups available
If error	nil	

ATME.C_Group.getReadyToBoardGroupsForCoalition(coalitionName, zoneName, allGroup)

Description : this method retrieves all groups ready to board for a specific coalition

Parameters :

coalitionName	string	coalition names : "RED", "BLUE" ou "NEUTRAL"
----------------------	--------	--

Return values:

Without error :	table	(ATME.C_Group) group table. Index are group names. Table will be nil if no groups available
If error	nil	

Object:changeReadyToBoard(value)

Description : This method set an infantry group ready to be embarked in a troop transport unit. This is a prerequisite for embarkation. The group is stopped if it is movement.

Parameters :

value	boolean	If true Group stand up ready to board, if false it will continues its mission.
--------------	---------	--

Return values: None

Object:activate()

Description : Activation of a group is required for groups defined with delayed activation. When activated, the group becomes visible. The group is already created in this case.

Parameters : None

Return values: None

Object:disable()

Description : This method disables a complete group. As a reminder, disabling a group removes it from the game without visible destruction and without DCS event S_EVENT_DEAD .

The following module handlers still enabled:

- **onDisableGroupHandler**
- **onDeleteGroupHandler** with isEnabled = false

Parameters : None

Return values: None

Object:freeze(on)

Description : This method freeze or not a group composed of AI units. As soon as fixed, the group will no longer make any action and will be uncontrolled by DCS.

Parameters :

on	boolean	If true, the group is turned off (frozen), if false it is checked again by DCS.
-----------	---------	---

Return values: None

Object:getBarycentre()

Description : return the group barycenter

Parameters : None

Return values:

	table	3D Point x, y et z
--	-------	--------------------

Object:getCategoryName()

Description : This method returns the group category name.

Parameters : None

Return values:

	string	category name : "AIRPLANE", "HELICOPTER", "GROUND" or "SHIP"
--	--------	---

Object:getClassName()

Description : retrieves the of the object ATME class name, in this case "C_Group". This method exists for all classes.

Parameters : None

Return values:

	string	Nom de la classe ATME class name
--	--------	----------------------------------

Object:getCoalitionName()

Description : recover the group coalition name

Parameters : None

Return values:

Without error :	string	coalition group name : "RED", "BLUE" or "NEUTRAL"
-----------------	--------	---

Object:getDCSGroup()

Description : This method is used to retrieve the group as managed by DCS (Group Class Object).

Parameters : None

Return values:

Without error :	table	DCS Group class instance
-----------------	-------	---------------------------------

Object:getFirstUnit()

Description : This method retrieves the first alive unit of a group.

Parameters : None

Return values:

table	C_AIUnit ou C_Player instance
-------	---

Object:getLastUnit()

Description : This method retrieves the last alive unit from a group.

Parameters : None

Return values:

table	C_AIUnit ou C_Player Instance
-------	---

Object:getMaxDistance(reference)

Description : calculates the maximum group distance from a reference. This is done the distance from the furthest alive unit. The calculation is done on the x, y and z axes.

Reference can be :

- A 2D Point or 3D,
- Unit AI position
- Player unit position
- Static object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y Ai Unit Type ATME.C_AIUnit Ai player unit type ATME.C_Player Static object type ATME.C_StaticObject
------------------	-------	---

Return values:

number	Distance m
--------	------------

Object: getMaxHDistance(reference)

Description calculates the maximum horizontal group distance from a reference. This is done the distance from the furthest alive unit. The calculation is done on the x and z axes.

Reference can be :

- A 2D Point or 3D,
- Unit AI position
- Player unit position
- Static object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y Ai Unit Type ATME.C_AIUnit Ai player unit type ATME.C_Player Static object type ATME.C_StaticObject
------------------	-------	---

Return values:

number	Distance m
--------	------------

Object: getMinDistance(reference)

Description : calculates the minimum group distance from a reference. This is done the distance from the nearest alive unit. The calculation is done on the x, y and z axes.

Reference can be :

- A 2D Point or 3D,

- Unit AI position
- Player unit position
- Static object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y Ai Unit Type ATME.C_AIUnit Ai player unit type ATME.C_Player Static object type ATME.C_StaticObject
------------------	-------	---

Return values:

number	Distance m
--------	------------

Object: getMinHDistance(reference)

Description : calculates the minimum horizontal group distance from a reference. This is done the distance from the nearest alive unit. The calculation is done on the x and z axes.

Reference can be :

- A 2D Point or 3D,
- Unit AI position
- Player unit position
- Static object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y Ai Unit Type ATME.C_AIUnit Ai player unit type ATME.C_Player Static object type ATME.C_StaticObject
------------------	-------	---

Return values:

number	Distance m
--------	------------

Object: getName()

Description : retrieves group Name

Parameters : None

Return values:

Without error :	string	Group Name
If error	nil	

Object:getNbUnits()

Description : retrieve number of alive unit

Parameters : None

Return values:

	number	number of alive unit
--	--------	----------------------

Object:getNextUnit()

Description : retrieve next alive unit. You must have initialized a scan of the group with getFirstUnit or getLastUnit.

Parameters : None

Return values:

	table	C_AIUnit ou C_Player instance
	nil	If no unit is found or scan in progress

Object:getPickupZone()

Description : retrieves a pickup zone for infantry troops

Parameters : None

Return values:

	string	DCS zone name
	nil	If no zone is linked to the group

Object:getPreviousUnit()

Description : retrieve previous alive unit in a group, you must have initialized a scan of the group with getFirstUnit or getLastUnit.

Parameters : None

Return values:

table	C_AIUnit ou C_Player instance
nil	If no unit is found or scan in progress

Object:getUnitByName(name)

Description : retrieve Unit in group with the name in parameters. Unit must be alive

Parameters : None

Return values:

table	C_AIUnit ou C_Player Instance
nil	If no Unit is found

Object:getUnits()

Description: retrieve a table with all alive unit from the group at a t time.

Parameters : None

Return values:

table	C_AIUnit or C_Player instance table
-------	---

Object:hasPickupZone()

Description : This method determines whether a DCS zone defining a boarding area has been assigned to an infantry group.

Parameters : None

Return values:

boolean	True if the infantry group has an associated boarding DCS zone. False otherwise.
---------	--

Object:isActivated()

Description : returns the activation state of a group defined as delayed activation in the mission editor.

Parameters : None

Return values: None

boolean	True if group is activated,else false .
---------	---

Important note:

Beware, the disable function destroys the group. Also, in this case, the group no longer exists for ATME after using this function.

Also, currently, the test uses a DCS function that is based on the unit.

Object:isBoardingStarted()

Description : This method is used to determine whether an infantry group is embarked in a troop transport unit.

Parameters : None

Return values:

boolean	True if the group is embarking,else false
---------	---

Object:isInDCSZone(zoneName, allGroup)

Description : test if the group is in a DCS zone defined in the mission editor. It is possible to know if the whole group or only a part of the group is inside

Parameters : None

zoneName	string	DCS zone Name
allGroup	boolean	If true all group must be inzone else part of a group

Return values:

Without error :	boolean	True / false
If error	nil	

Object:isInZone2D(reference, radius, allGroup)

Description : This method determines whether the AI unit is in a horizontal zone defined by a Unit a circle (center radius). It is possible to know if the whole group or only a part of the group are inzone.

Reference can be :

- A 2D Point or3D,
- Unit AI position
- Player unit position
- Static object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit. Type ATME.C_Player Static Object Type ATME.C_StaticObject
radius	number	Circle radius in meter
allGroup	boolean	If true check that all the group is in zone else part of the group.

Return values:

Without error :	boolean	True / false
If error	nil	

Object:isInZone3D(reference, radius, allGroup)

Description : This method determines whether the group is in a spherical zone defined by a Unit a sphere (center radius). The center can be :

- 2D Point or 3D point,
- A AI Unit position
- A player Unit position
- A static Object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject
radius	number	Spherical radius in meter
allGroup	boolean	If true check that all the group is in zone else part of the group.

Return values:

Without error :	boolean	True / false
If error	nil	

Object:isNear(reference, radius, deltaAltitude, allGroup)

Description : This method test if a reference is close to the group. The zone is defined by a cylinder represented by a horizontal circle, the two sides of the cylinder correspond to an altitude difference (the group is in the center).

Center can be :

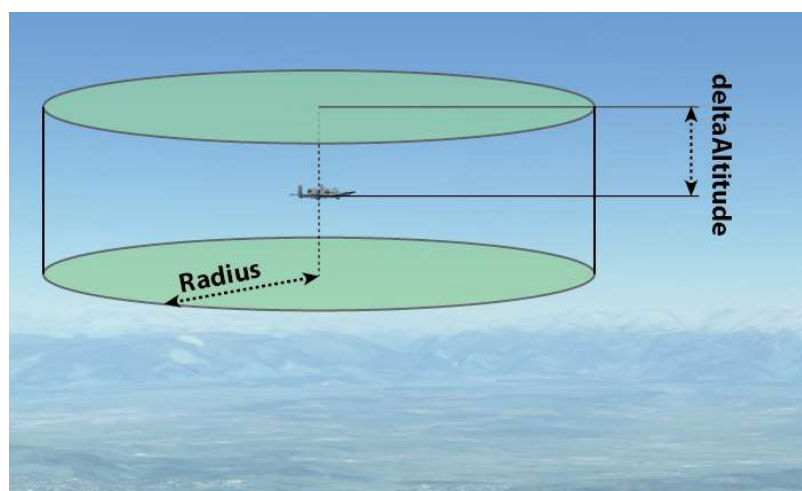
- 2D Point or 3D point,
- A AI Unit position
- A player Unit position
- A static Object position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static object Type ATME.C_StaticObject
radius	number	Circle radius top and bottom in meters
deltaAltitude	number	Altitude difference between top/bottom and center m
allGroup	boolean	If true check that all the group is in zone else part of the group.

Return values:

Without error :	boolean	True / false
If error	nil	



Object:isOnlyInfantry()

Description : check if group contains only infantry units at a T time

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isOnlyAI()

Description : check if group contains only AI units at a T time

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isReadyToBoard()

Description : test if an infantry group is ready to be embarked in a troop transport unit. This is a prerequisite for embarkation. The group is also stopped if it was movement.

Parameters : None

Return values:

boolean	True if group is ready to board else false
---------	--

Object:isSignalSet()

Description : test if a group is actually tested on flight hover monitoring

Parameters : None

Return values:

boolean True /false

Object:isStopped()

Description : This method test status of a group in movement or stopped. Its state depends on the use of the move or stop functions.

Parameters : None

Return values:

boolean True if group is stopped else false
--

Object:move()

Description : This method is used to route a group of AI units after holding in position

Parameters : None

Return values: None

Object:resetPickupZone()

Description : This method removes the link of an infantry group to a DCS boarding area. The group does not change state ready to be on board

Parameters : None

Return values: None

Object:resetSignal()

Description : This method removes zone monitoring. As a result, the group will no longer signals overflight by a troop transport unit.

Parameters : None

Return values: None

Object:setPickupZone(zoneName, random)

Description : This method allows an infantry group to be linked to a boarding pick up zone. This field is defined by a DCS zone called zoneName defined in the mission editor. The random parameter makes it possible to put a random stop of the group and its transition to the state ready to embark.

Once the group has been assigned to a boarding pickup zone, if it enters the zone it will be set “ready to board” randomly. If the group exit the pickupzone while it is still not ready to board, it will be immediately put into this state. Everything is managed automatically. Several groups may have the same boarding pick up zone.

A boarding pick up zones affects signal of a ready-to-board infantry group. In this case, only one group of the zone will signal itself. refers to the setSignal function .

Parameters :

zoneName	string	DCS zone name
random	number	number between 0 and 1 : 1 : group is ready to board once inzone 0 : group is never ready to board except when it exit the zone

Return values: None

Object:setRoute(...)

Description : This method allows to associate all or part of the waypoints of another group defined in the mission. The Parameters for this function are variable and this method allows to associate all or part of the waypoints of another group defined in the mission. The Parameters for this function are variable and listed below. It is possible to change the route of any group, or even to ask a group to reconstruct its own waypoints.

Parameters case 1 : waypoints used are those of the group as defined in the mission editor. If the group is a copy of an initial group (spawn), the waypoints will be those of the initial group.

Parameters case 2 : waypoints used are those of the group whose name are in parameters. This name must exist in the mission editor. The second parameter allows to set from where the waypoints are resumed:

- Waypoint number: Previous waypoints will be ignored.
- A waypoint name: If it exists, the previous waypoints will be ignored. If it does not exist, only the last waypoint will be resumed. A reference: All waypoints defined after the waypoint closest to the reference will be resumed. A reference can be :
 - 2D Point or 3D point,
 - Ai Unit Position
 - Player Unit Position
 - Static Object Position.

groupName	string	Group Name used in Mission editor
reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject

Return values: None

Object:setSignal(radius, signalType)

Description : set Monitoring. As soon as an aerial transport unit (currently only helicopters) begins the overflight of the zone whose center is the group first unit alive, it triggers a signal. If the group is linked to a boarding pickup zone where other groups are present, only one of them will trigger the signal. Case of destruction of the group, another group (if any) will immediately take over. In this case, and if the signal is a smoke, there may be several smoke signal active simultaneously, from the destroyed groups and from the group always active. group sends the signals at regular intervals as long as a troop transport unit is flying over :

- If the signal is a flare, the shot is regular with however a random factor of time
- If the signal is a smoke, the smoke will be regenerated regularly with a random time interval.

resetSignal function will stop monitoring and transmitting signals.

Parameters :

radius	number	Monitoring zone radius
signalType	string	Special string see below

Return values: None

signalType parameters values :

- String is set with 2 words : "**TypeSignal Color**".
- "**TypeSignal**" is "**SIGNAL_FLARE**" for a flare and "**SIGNAL_SMOKE**" for a smoke.
- "**Color**" must be one allowed colours "**RANDOM**" can also be used. See **ATME.C_Flare** or **ATME.C_Smoke** for more informations.
- Exemples :
 - "**SIGNAL_SMOKE BLUE**" will trigger a blue smoke
 - "**SIGNAL_FLARE RANDOM**" will trigger a flare with random color

Object:stop()

Description : This method allows position stand up of the Ai Units group.

Parameters : None

Return values: None

ATME. C_GroupSpawnDatas class

ATME.C_GroupSpawnDatas.duplicateFromMissionDatas(groupName, newGroupName)

Description : This function creates an instance to retrieve the data needed to create a new group regards to a defined group in the mission editor.

Parameters :

groupName	string	Group name as defined in mission
newGroupName	string	New group Name

Return values:

Without error :	table	New data instance.
Si erreur	nil	

Object:getClassName()

Description : This method retrieves the Object ATME class name, in this case "C_GroupSpawnDatas". This method exists for all classes.

Parameters : None

Return values:

	string	ATME class Name
--	--------	-----------------

Object:getName()

Description : retrieves the name of the data group that corresponds to the new name

Parameters : None

Return values:

Without error :	string	Group Name
If error	nil	

Object:spawn(...)

Description : This method creates a new group of units from the data present in the current instance. The new group name must not yet be used at the time of creation. The same applies to the new group units whose name has been modified (see below). This function can have several variable Parameters, described below. These Parameters concern the use of the waypoints of the initial group or of another group as well as the new group position. This function triggers the following handlers in the user modules :

- callback **onCreateGroupHandler**
- for each New Unit from new group callback **onCreateAIUnitHandler**

New Units will be called like that :

- Origin Name Preceded by "**SpawnUXXXXX**", **XXXXX** is unique number preceded if necessary by 0.

Parameters case 1 : waypoints used are those of the group as defined in the mission editor. The position of the new group and that of the initial group.

Parameters case 2 : The position is specified. The waypoints used are those of the initial group used during the **duplicateFromMissionDatas**.

point	table	3D Point x, y et z group position
--------------	-------	-----------------------------------

Parameters case 3 : The position must be specified. The waypoints used are those of the group whose name is in parameter (it can be the same name as the initial group). All waypoints are resumed.

point	table	3D Point x, y et z group position
groupNameWP	string	Name of waypoints source group to resume

Parameters cas 4 : The position must be specified. The waypoints used are those of the group whose name is in parameters. This name must exist in the mission editor. The third parameter allows to set from where the waypoints are resumed:

- Waypoint number: Previous waypoints will be ignored.
- A waypoint name: If it exists, the previous waypoints will be ignored. If it does not exist, only the last waypoint will be resumed. A reference: All waypoints defined after the waypoint closest to the reference will be resumed. A reference can be
 - A 2D Point or 3D Point,
 - Ai unit Position
 - Player Unit position
 - Static object position.

point	table	Point 3D x, y et z group position
groupNameWP	string	Name of waypoints source group to resume
reference	table	3D Point x, y et z, or 2D Point x,y Ai Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject

Return values: None

ATME.C_IndexList class

This class allows an indexed list that can contain nil values. The index will necessarily be numeric. If an entry is deleted, the index of the following entries does not change.

ATME.C_IndexList()

Description : This function creates a new list

Parameters : None

Return values:

Without error	table	C_IndexList new instance
Si erreur	nil	

Object:add(item)

Description : This method adds an item to the list.

Parameters :

item	table	Item added
-------------	-------	------------

Return values: None

Object:get(index)

Description : This method is used to retrieve the item positioned at the parameter index.

Parameters :

index	number	Index of the item to retrieve
--------------	--------	-------------------------------

Return values:

	table	Item at index position
--	-------	------------------------

Object:getName()

Description : This method retrieves the name of the ATME class of the object, in this case "index list". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getCount()

Description : retrieves the number of items in the list.

Parameters : None

Return values:

number	number of items in the list.
--------	------------------------------

Object:remove(index)

Description : This method is used to delete the element present at the index in parameter.

Parameters :

index	number	Index item to delete
--------------	--------	----------------------

Return values: None

Object:removeAll()

Description : This method removes all items from the list.

Parameters : None

Return values: None

Object:pairs()

Description : This method redefines the lua method to scan entries in a list. It is used in a loop.

Parameters : None

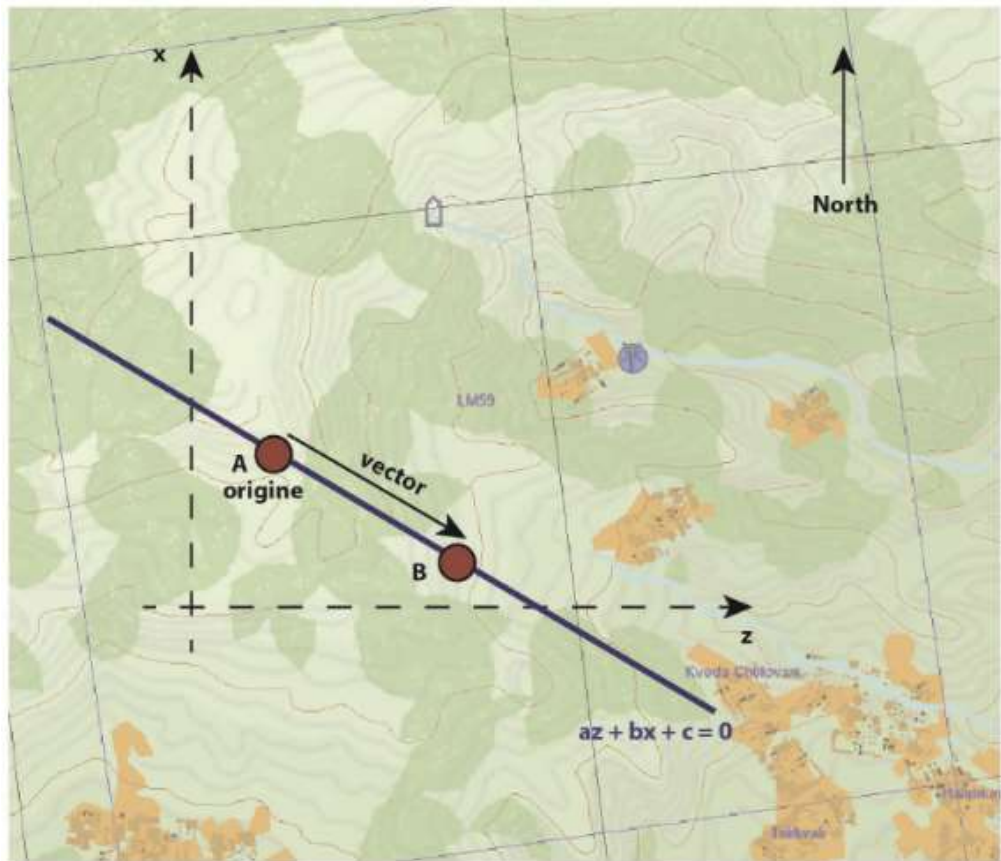
Return values: linked with pairs (lua)

ATME.C_Line2D class

This class allows to manage straight lines on the horizontal plane of DCS defined by the x and z axes.

The format is: $az + bx + c = 0$.

A straight line also has a point of origin, named origin below.



ATME.C_Line2D(...)

Description : This function creates a line on the horizontal plane on the x and z axes.

Parameters case 1 : Creates copy of another line

line	table	C_Line2D instance
------	-------	-------------------

Parameters case 2 : Creates a line from two points A and B, based on the x and z coordinates. Point A is the origin of the line.

pA	table	Point A -3D Point x, y et z, or 2D Point x,y
pB	table	Point B -3D Point x, y et z, or 2D Point x,y

Parameters case 3 : Creates a line from a vector and a point that will be the origin, based on the x and z coordinates.

vector	table	C_Vector3D Instance
origine	table	3D Point x, y et z, or 2D Point x,y

Return values:

Without error	table	C_Line2D new instance
Si erreur	nil	

Object:get()

Description : recover the coordinates a, b and c of a horizontal line on the x and z axes.

Parameters : None

Return values:

number	a
number	b
number	c

Object:getA()

Description : This method is used to retrieve a.

Parameters : None

Return values:

number	a
--------	---

Object:getB()

Description : This method is used to retrieve b

Parameters : None

Return values:

number	b
--------	---

Object:getC()

Description : This method is used to retrieve c

Parameters : None

Return values:

number	c
--------	---

Object:getName()

Description : retrieves the object ATME class name, in this case "C_Line2D". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getOrigine()

Description : recover the origin point of the horizontal straight line

Parameters : None

Return values:

table	3D Point x, y et z représent origin
-------	-------------------------------------

Object: getPerpendicular(offset)

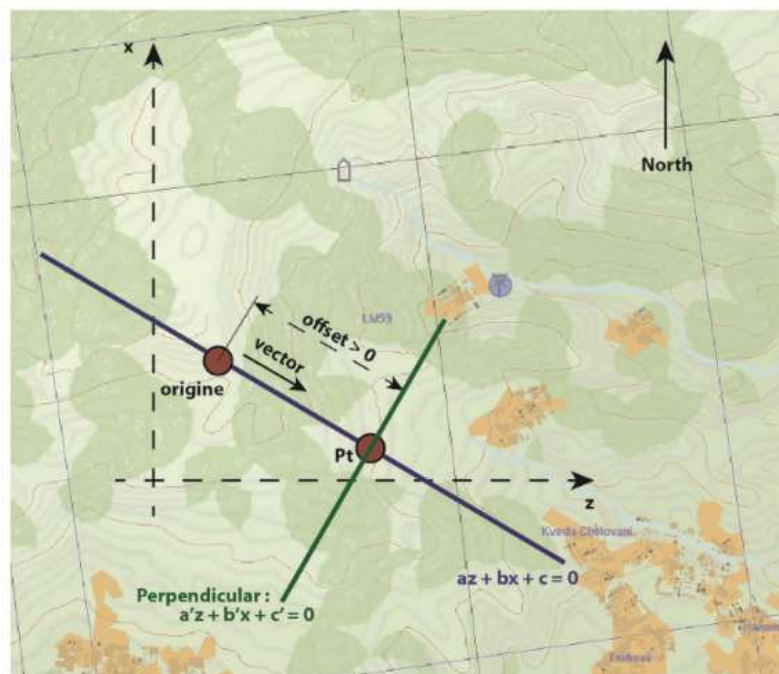
Description : recover the perpendicular line passing at the offset point with respect to the origin. The point of origin of this new line will be the point of intersection (Pt in the diagram). If offset is negative, the point will be the opposite of the example below for the same given value.

Parameters :

offset	number	Offset respect to origin point
---------------	--------	--------------------------------

Return values:

table	C_Line2D Instance
-------	-------------------



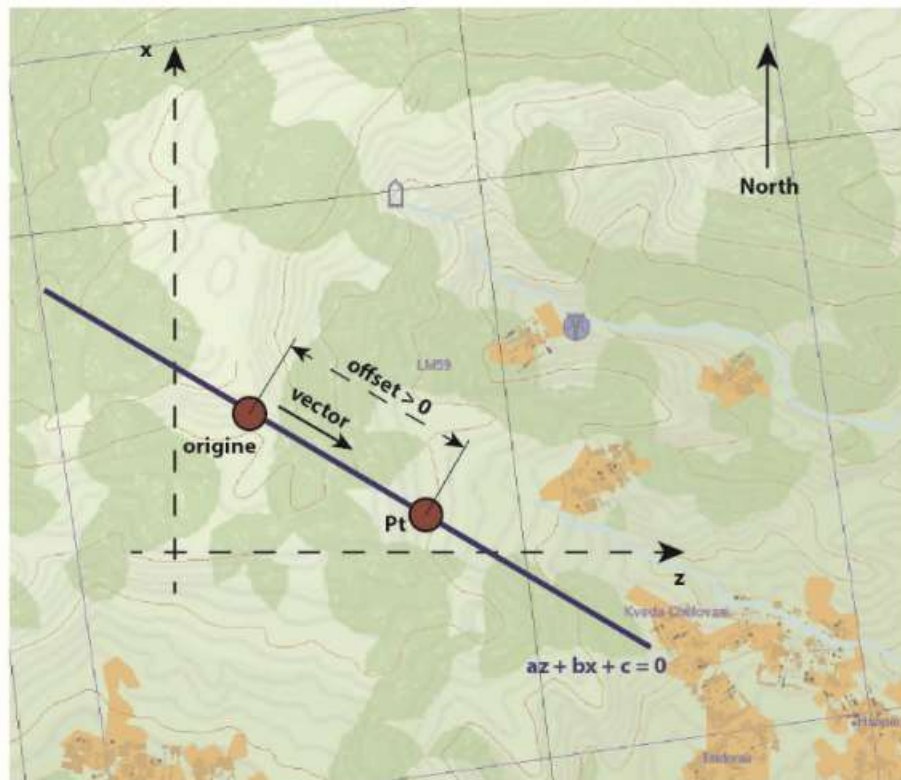
Object: getPointFromOrigine(offset)

Description : retrieves the point on the right offset from the origin. Caution, offset can be positive or negative depending on the half-line concerned in relation to the origin. The sign depends on the direction vector of the line; In the same sense, offset will be positive

Parameters :

offset	number	Offset respect to origin point
---------------	--------	--------------------------------

Return values:



Object:getX()

Description : calculate x with known z (if possible).

Parameters : None

Return values:

number	Calculated x or nil if impossible
--------	-----------------------------------

Object:getZ()

Description : calculate z with x known (if possible).

Parameters : None

Return values:

number	Calculated coordinate z or nil if impossible
--------	--

Object:isNSAxis()

Description : test if the line is parallel to the axis NS (coordinate x for DCS)

Parameters : None

Return values:

boolean True if the line is parallel, else false

Object:isWEAxis()

Description : test if line is parallel to the WE axis (z coordinate for DCS)Parameters : None

Return values:

boolean True if the line is parallel, else false

ATME.C_MenuF10 Class

This class manages the player menus. It deals with the display or not of the menu items and their classification on the display. The classification on the display is determined by the functions **ATME.setF10groupIDAlphaOrder()** et **ATME.setF10AlphaOrder()**.

By default, this is an alphabetical order.

ATME.C_MenuF10(player, label, parent)

Description : This function creates a new menu accessible from F10 and linked with a player. The menu, displayed with the label, is linked to a parent menu; It is the basic menu (root) if parent menu is nil.

Parameters :

player	table	C_Player Instance
label	string	Menu label
parent	table	C_F10Menu Instance parent from menu to create

Return values: None

Object:append(groupId, menuLabel, radioHandler, argsRadioHandler)

Description : This method adds an entry to the desired F10 menu. The groupId is used to identify entries in the set of menus. It acts on the classification of the display if the option has been activated by the function nil. **ATME.setF10groupIDAlphaOrder()**. This value is also useful for deleting one or more entries. The groupId can be the same for multiple entries.

The menu will only be displayed if there is at least one entry, including its submenus.

Parameters :

groupId	number	An integer to identify one or more menu entries F10.
menuLabel	string	Text displayed in menu F10
radioHandler		Function Handler called when a choice is done in a Radio menu
argsRadioHandler	table autre	Parameters set through menu F10 handler (table or other variable type)

Return values: None

Object:remove(groupId)

Description : This method removes all radio command entries corresponding to the specified groupId. Deleting also affects the associated submenus.

Parameters :

groupId	number	Integer to identify one or more F10 menu entries.
----------------	--------	---

Return values: None

Object:removeAll()

Description : This method removes all entries from the F10 menu and its submenus.

Parameters : None

Return values: None

ATME.C_Module Class

ATME.C_Module(name, handlers, debugOn)

Description : This function declares and creates a new user module. This module must be created before the **ATME.run** function is launched in the mission editor. The module name must be unique.

The handlers defined in the new module will be memorized by ATME Core and called when required.

DebugOn is used to indicate whether or not the module is in debug mode.

Parameters :

name	string	Module Name
handlers	table	Handler list defined in the new module
debugOn	boolean	True if module is in debug mode else false

Return values:

Without error	table	C_Module new instance
Si erreur	nil	

Handlers linked with players :

- **onCreatePlayerHandler(player)** : Called by **ATME Core** when creating a new player. The player parameter corresponds to the new player's **C_Player** instance
- **onDeletePlayerHandler(player)** : Called by **ATME Core** when a player is destroyed (or deactivated). The player parameter is the player's **C_Player** instance.
- **onUpdatePlayerHandler(player, events)** : Called by **ATME Core** every second for all active players. The player parameter is the player's **C_Player** instance. Events is the list of active events at the time of the call (see class **C_EventMgr**).
- **onTakeoffPlayerHandler(player)** : Called by **ATME Core** when the player takes off. The player parameter is the player's **C_Player** instance.
- **onLandingPlayerHandler(player)** : Called by **ATME Core** when the player land. The player parameter is the player's **C_Player** instance.
- **onStartEnginePlayerHandler(player)** : Called by **ATME Core** when the player engines start. The player parameter is the player's **C_Player** instance.
- **onStopEnginePlayerHandler(player)** : Called by **ATME Core** when the player engine stop. The player parameter is the player's **C_Player** instance.

Handlers linked with AI Unit :

- **onCreateAIUnitHandler(AIUnit)** : Called by **ATME Core** when creating a new AI unit. The **AIUnit** parameter corresponds to the **C_AIUnit** instance of the new AI unit.
- **onDeleteAIUnitHandler(AIUnit, isEnabled)** : Called by **ATME Core** during the destruction (or deactivation) of an AI unit. The **AIUnit** parameter corresponds to the **C_AIUnit** instance of the AI unit. **isEnabled** is true if the AI device is active, false if it is disabled.
- **onDisableAIUnitHandler(AIUnit)** : Called by **ATME Core** when deactivating an AI unit. The **AIUnit** parameter corresponds to the **C_AIUnit** instance of the AI unit. The handler **onDeleteAIUnitHandler** will then be called with the parameter **isEnabled** to false.
- **onTakeoffAIUnitHandler(AIUnit)** : Called by **ATME Core** when the AI unit takes off. The **AIUnit** parameter corresponds to the **C_AIUnit** instance of the AI unit.
- **onLandingAIUnitHandler(AIUnit)** : Called by **ATME Core** when the AI unit lands. The **AIUnit** parameter corresponds to the **C_AIUnit** instance of the AI unit.
- **onStartEngineAIUnitHandler(AIUnit)** : Called by **ATME Core** when the AI unit engines start. The **AIUnit** parameter corresponds to the **C_AIUnit** instance of the AI unit.
- **onStopEngineAIUnitHandler(AIUnit)** : Called by **ATME Core** when the AI unit stop engines. The **AIUnit** parameter corresponds to the **C_AIUnit** instance of the AI unit.

Handlers linked to groups :

- **onCreateGroupHandler(group)** : Called by **ATME Core** when creating a new group. The **group** parameter matches the new player's **C_Group** instance
- **onDeleteGroupHandler(group, isEnabled)** : Called by **ATME Core** when a group is destroyed (or deactivated). The **group** parameter is the player's **C_Group** instance. **isEnabled** is true if the group is active, false if it is disabled.
- **onDisableGroupHandler(group)** : Called by **ATME Core** when disabling a group. The **group** parameter is the **C_Group** instance of the group. The handler **onDeleteGroupHandler** will then be called with the **isEnabled** parameter at false.

Handlers liés aux objets statiques :

- **onCreateStaticObjectHandler(object)** : Called by **ATME Core** when creating a new static object. The object parameter corresponds to the **C_StaticObject** instance of the new static object.
- **onDeleteStaticObjectHandler(object)** : Called by **ATME Core** when a static object is destroyed. The object parameter is the **C_StaticObject** instance of the static object.

General Handlers :

- **onTimerHandler(events)** : Called by **ATME core** once per second. Events is the list of active events at the time of the call (see class **C_EventMgr**).
- **onModuleStartHandler(initOk)** : Called twice by **ATME Core** when starting ATME, in the **ATME.run** function. The **initOk** parameter takes the following two successive values:
 - At first call, **initOk** is false. AI and player units have not been initialized at this stage.
 - At **second call**, **initOk** is true. AI and player units are initialized.

Global variables and extension of object C_AIUnit, C_Player, C_Group or C_StaticObject instances type :

When creating a module, a module-specific entry is created in the **ATME.modules** table. The name of this entry is **ATME.modules ["module name"]**. This entry can be used freely by the module designer to declare data or functions useful to other modules or to be accessible from the mission editor.

In addition, the classes **C_AIUnit**, **C_Player**, **C_Group** and **C_StaticObject** also have a new field dedicated to the module, **modules ["module name"]**.

Object:error(text)

Description : This function generates an error message that will be displayed in a window and will block the script. Note that all error messages are stored in dcs.log and prefixed with "[ATME-ModuleName]".

Parameters :

text	string	Error message display text
-------------	--------	----------------------------

Return values: None

Object:getClassName()

Description : This method allows to retrieve the ATME class name of an object, in this case "**C_Module**". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:isDebugEnabled()

Description : return if a module is debug mode or not.

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:output(text, level)

Description : This function displays a debug message at a defined level and saved in dcs.log. All debug messages are prefixed with "[ATME-ModuleName]". The ATME.setDebugLevel is used to set the maximum level of messages displayed. This function can be used directly by a script in the mission editor.

debug message is displayed only if the debug mode has been activated for the module concerned by the method **C_Module.setDebug**

Parameters :

text	string	debug message text
level	number	Debug level to be displayed

Return values: None

Object:createAbsoluteUserTrigger(name, condition)

Description : creates a new user trigger based on a time condition related to mission time. A user trigger is identified by its name and always linked to the module to avoid duplicates. The trigger will be deleted automatically after it is deactivated.

Parameters :

name	string	User trigger name
condition	string	Time condition of activation for the user trigger

Return values:

Without error :	table	C_UserTrigger instance
If error	nil	

Explanation of the **condition** parameter:

Condition is a string defining the triggering and stopping of a second trigger.

The format is strict and follows the rule "(+)XXX" or "(+)XXX (+)YYY"

- "(+)XXX" Indicates a trigger that will only trigger once on time **XXX**, from mission start. The absence of the + means from mission start time, otherwise relative to the creation of the trigger if + is indicated.
- "(+)XXX (+)YYY" is for a trigger which will trigger in a defined period behind **XXX** and **YYY**. for **XXX**, relative to mission start time (if no +), relative to trigger creation(if + is

indicated). for **YYY**, relative to mission start time (if no +), relative to **XXX** if + is indicated (**XXX+YYY** secondes)

"**XXX**" and "**YYY**" are used :

- Positif integer (number of seconds),
- **HH:MM:SS** form represents hours minute seconds.

Exemples :

- "**10**" The trigger will trigger once, 10 seconds mission start
- "**+4200**" same case
- "**10 20**" will trigger over the period of 10 seconds to 20 seconds after the mission start.
- "**00:10:00 +20**" will be triggered over the period of 10 minutes after the mission start over a period of 20 seconds.
- "**600 +20**" same case.
- "**600 +00:20:00**" The trigger will trigger over the period of 10 minutes to 30 minutes after the start of the mission
- "**+00:00:10 20**" same case
- "**+00:10:00 +1805**" same case

Object:createFlagRelativeUserTrigger(name, flagNumber, condition)

Description : creates a new user trigger based on a time condition associated with the activation of a flag. A user trigger is identified by its name and always linked to the module calling this function to avoid duplicates. The user trigger will be deleted automatically after it is deactivated.

Parameters :

name	string	User trigger Name
flagNumber	number	Flag number wich will lauch the time period
condition	string	User trigger time condition

Return values:

Without error :	table	C_UserTrigger Instance
If error	nil	

Explanation of the **condition** parameter:

Condition is a string defining the triggering and stopping of a second trigger.

The format is strict and follows the rule **"(+)**XXX**"** or **"(+)**XXX** (+)**YYY**"**

- **"(+)**XXX**"** Indicates a trigger that will only trigger once on time **XXX**, from mission start. The absence of the + means from mission start time, otherwise relative to the creation of the trigger if + is indicated.
- **"(+)**XXX** (+)**YYY**"** is for a trigger which will trigger in a defined period behind **XXX** and **YYY**. for **XXX**, relative to mission start time (if no +), relative to trigger creation(if + is indicated). for **YYY**, relative to mission start time (if no +), relative to **XXX** if + is indicated (**XXX+YYY** secondes)

"XXX" and **"YYY"** are used :

- Positif integer (number of seconds),
- **HH:MM:SS** form represents hours minute seconds.

Exemples :

- **"10"** The trigger will trigger once, 10 seconds after flag activation
- **"**+10**"** same case
- **"4200"** same case
- **"10 20"** trigger will trigger from 10 s after flag activation to 20s after mission start. Be aware when the flag is active The period may have an end before its beginning.
- **"**+10 20**"** same case
- **"00:10:00 **+20**"** will trigger over the period of 10 minutes after flag activation for a duration of 20 seconds.
- **"**+600 +20**"** same case
- **"600 **+00:20:00**"** will trigger over the period of 10 minutes to 30 minutes after the flag is activated
- **"**+00:10:00 +1805**"** same case

Object:getUserTriggerByName(name)

Description : This method returns a scheduled and / or activated user trigger.

Parameters :

name	string	User trigger name
-------------	--------	-------------------

Return values:

Without error :	table	C_UserTrigger name
If error	nil	

Object:removeUserTrigger(name)

Description : This method removes a scheduled and / or activated user trigger. If it is activated, it is immediately inactive.

Parameters :

name	string	User trigger name
-------------	--------	-------------------

Return values: None

Object:userTriggerExists(name)

Description : Test if a user trigger exists, that is to say, that it is programmed and / or activated.

Parameters :

name	string	User trigger Name
-------------	--------	-------------------

Return values:

Without error :	boolean	True if au user trigger exists
If error	nil	

ATME. C_Player class

The instances of this class are created and deleted by ATME Core.

ATME.C_Player.exists(name)

Description : Function checking if a player unit is alive at T time

.Parameters :

name	string	Player Unit
-------------	--------	-------------

Return values:

Without error :	boolean	True if C_Player instance exist else false
If error	nil	

ATME.C_Player.getByName(name)

Description : Function to retrieve, from its name, the player unit alive at a T time

.Parameters :

name	string	Player unit Name
-------------	--------	------------------

Return values:

Without error :	table	C_Player Instance
If error	nil	

Object:crossAxisFromLeftToRight(pA, pB)

Description : method verifies the player's crossing of the horizontal line (2D straight line on the x, z axes) represented by the points A and B. The direction AB is oriented and makes it possible to define the direction of the crossing, here on the left to the right.

Parameters :

pA	table	3D Point x, y et z, or 2D Point x,y
pB	table	3D Point x, y et z, or 2D Point x,y

Return values:

Without error :	boolean	True if crossed from left to right.
If error	nil	

Object:crossAxisFromRightToLeft(pA, pB)

Description : This method verifies the player's crossing of the horizontal line (straight line 2D on the x, z axes) represented by the points A and B. The direction AB is oriented and makes it possible to define the direction of the crossing, here on the right to the left.

Parameters :

pA	table	3D Point x, y et z, or 2D Point x,y
pB	table	3D Point x, y et z, or 2D Point x,y

Return values:

Without error :	boolean	True if crossed from right to left
If error	nil	

Object:display(text, duration)

Description : This method displays a message for a specific player. The message will be displayed for the duration indicated.

Parameters :

text	string	Message text to display
duration	number	Display message duration

Return values: None

Object:explode(power)

Description This method triggers an explosion of force defined on a player's unit.

Parameters :

power	number	Explosion power
--------------	--------	-----------------

Return values: None

Object:getAGLAltitude()

Description : This method retrieves the meter altitude of a player's unit above ground level

Parameters : None

Return values:

Without error :	number	Altitude m
-----------------	--------	------------

Object:getAzimuth()

Description : This method retrieves the azimuth (relative to the north) of a player's unit. This corresponds to the true heading without taking into account the magnetic variation.

Parameters : None

Return values:

Without error :	number	Player unit Azimuth in degrees (0-360)
-----------------	--------	--

Object:getCallsign()

Description : This method retrieves the name of the coalition from the AI unit. The callsign consists of « name id1-id2 », exemple « enfield 1-1 »

Parameters : None

Return values:

	string	callsign Name
	string	callsign Id1
	string	callsign Id2

Object:getClassName()

Description : retrieves the name of the object ATME class of the object, in this case "C_Player". This method exists for all classes.

Parameters : None

Return values:

	string	ATME class name
--	--------	-----------------

Object: getCoalitionName()

Description : retrieves the name of the coalition from a player's unit.

Parameters : None

Return values:

Without error :	string	Coalition player's name : "RED", "BLUE" ou "NEUTRAL"
-----------------	--------	--

Object: getCountryName()

Description : retrieve the country name of a player's unit.

Parameters : None

Return values:

Without error :	string	Player's country name
-----------------	--------	-----------------------

Country names (from DCS)

"RUSSIA", "UKRAINE", "USA", "TURKEY", "UK", "FRANCE", "GERMANY", "AGGRESSORS", "CANADA", "SPAIN", "THE_NETHERLANDS", "BELGIUM", "NORWAY", "DENMARK", "ISRAEL", "GEORGIA", "INSURGENTS", "ABKHAZIA", "SOUTH_OSETIA", "ITALY", "AUSTRALIA", "SWITZERLAND", "AUSTRIA", "BELARUS", "BULGARIA", "CHEZH_REPUBLIC", "CHINA", "CROATIA", "EGYPT", "FINLAND", "GREECE", "HUNGARY", "INDIA", "IRAN", "IRAQ", "JAPAN", "KAZAKHSTAN", "NORTH_KOREA", "PAKISTAN", "POLAND", "ROMANIA", "SAUDI_ARABIA", "SERBIA", "SLOVAKIA", "SOUTH_KOREA", "SWEDEN", "SYRIA"

Object:getDCSUnit()

Description : recovers a player's unit as managed by DCS (Unit Class Object).

Parameters : None

Return values:

Without error :	table	DCS Unit class instance
-----------------	-------	--------------------------------

Object:getF10MenuRoot()

Description : method returns the root menu of the player (menu F10 base).

Parameters : None

Return values:

	table	Menu root type ATME.C_F10Menu
--	-------	--------------------------------------

Object:getFuelRatio()

Description : This method allows to know the ratio of fuel remaining to full.Parameters : None

Return values:

	number	Ratio fuel remaining
--	--------	----------------------

Object:getGroup()

Description : This method retrieves the C_Group instance corresponding to a player's unit.
Parameters : None

Return values:

Without error :	table	C_Group instance linked with Player Unit
-----------------	-------	---

Object:getGroupsNameOnBoard()

Description : This method retrieves the list of infantry group names embedded in the player's unit. The unit of the player must be an authorized vehicle, that is to say have a capacity of transport of troops (personnel carrier).

Parameters : None

Return values:

	table	table with groups name on board. First group is at index 1
--	-------	--

Object:getHSpeed()

Description : recover the horizontal speed of the unit of a player m / s (calculated on the 2 axes x and z).

Parameters : None

Return values:

Without error :	number	Player Unit horizontal speed in m/s
-----------------	--------	-------------------------------------

Object: getLife()

Description : This method retrieves the "life" information from an AI unit.

Parameters : None

Return values:

number	Unit Life
--------	-----------

Object: getLifeRatio()

Description : This method allows to recover the ratio of life. This ratio is calculated from the life of the unit to its creation and its current state.

Parameters : None

Return values:

number	Life Ratio
--------	------------

Object: getMSLAltitude()

Description : retrieve the player Unit altitude in meters above sea level.

Parameters : None

Return values:

Without error :	number	Altitude m
-----------------	--------	------------

Object:getName()

Description : This method retrieves the Player Unit name.

Parameters : None

Return values:

Without error :	string	Player Unit Name
-----------------	--------	------------------

Object:getNbUnitsOnBoard()

Description : recovers the number of infantry on board.

Parameters : None

Return values:

number	number of infantry on board.
--------	------------------------------

Object:getNearestReadyToBoardGroups(radius)

Description : search for the nearest embarked infantry group within the specified radius. If no unit is found, nil will be returned.

Parameters :

radius	number	Search radius
---------------	--------	---------------

Return values:

table	Nearest infantry group ready to board (ATME.C_Group)
nil	If no group found

Object:getPosition()

Description : retrieves the position of the player unit (3D Point).

Parameters : None

Return values:

Without error :	table	Point 3D x, y et z player Unit position
-----------------	-------	---

Object:getPseudo()

Description : retrieve the nickname of a player, useful in multiplayer. The nickname is the login name. in single player, this pseudo is fixed

Parameters : None

Return values:

Without error :	string	Player Nickname
-----------------	--------	-----------------

Object:getRollAxisVector()

Description : returns a vector corresponding to the longitudinal axis of the AI unit, facing forward.

Parameters : None

Return values:

table	ATME.C_Vector3D vector
-------	------------------------

Object: getSpeed()

Description : recover the player unit speed m / s (calculated on the 3 axes).

Parameters : None

Return values:

Without error :	number	Player Unit speed in m/s
-----------------	--------	--------------------------

Object: getSpeedAzimuth()

Description : recover the true route (heading of the velocity vector on the x and z axes) of a player's unit.

Parameters : None

Return values:

Without error :	number	Player heading Route in degrees (0-360)
-----------------	--------	---

Object: getTypeName()

Description : retrieves the player's unit type name

Parameters : None

Return values:

Without error :	string	player's unit type name
-----------------	--------	-------------------------

Object: getVelocityVector()

Description : recover the speed vector of the player's unit

Parameters : None

Return values:

Without error :	table	Speed C_Vector3D Instance
-----------------	-------	---------------------------

Object: getVSpeed()

Description : This method retrieves the player's unit vertical speed m / s (taken on the y-axis).

Parameters : None

Return values:

Without error :	number	player's unit vertical speed m/s
-----------------	--------	----------------------------------

Object: inAir()

Description : test if player's unit is in Air.

Parameters : None

Return values:

Without error :	boolean	True / false
-----------------	---------	--------------

Object:isEngineStarted()

Description : test if player's Unit engine is started

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isGroundVehicle()

Description : test if Player's Unit is a ground vehicle

Parameters : None

Return values:

boolean	True / false
---------	--------------

Object:isHelicopter()

Description : test if Player's Unit is an helicopter.

Parameters : None

Return values:

Without error :	boolean	True / false
-----------------	---------	--------------

Object:isInDCSZone(zoneName)

Description : test if player's unit is in a DCS zone defined in the mission editor.

Parameters : None

zoneName	string	DCS zone name
-----------------	--------	---------------

Return values:

Without error :	boolean	True / false
If error	nil	

Object:isRouteInDirection(reference)

Description : determines if the player's unit is in the correct direction relative to a given reference.

A reference can be :

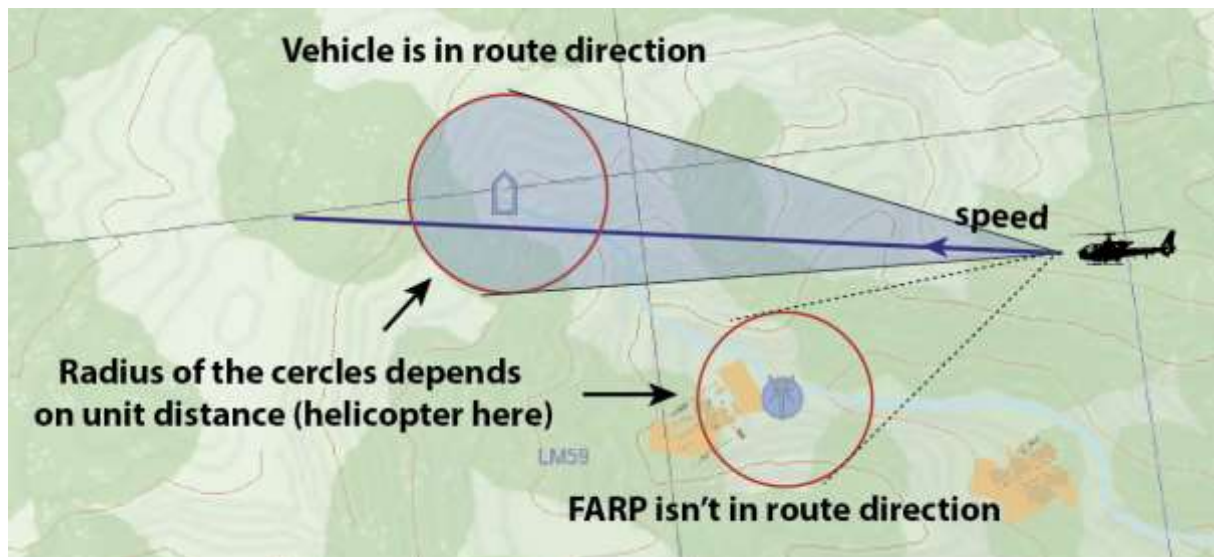
- A 2D Point or 3D Point ,
- Ai Unit Position
- Player Unit Position
- Static Object Position.

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject
------------------	-------	--

Return values:

Without error :	boolean	True if in correct direction / false
If error	nil	



Object:isInZone2D(reference, radius)

Description : test if player's unit is in a horizontal zone defined by a circle and a radius from :

- A 2D Point or 3D point ,
- Ai Unit Position
- Player Unit Position
- Static Unit position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject
radius	number	Radius from circle.

Return values:

Without error :	boolean	True if in zone else false
If error	nil	

Object:isInZone3D(reference, radius)

Description : test if player's unit is in a spherical zone defined by a sphere and a radius from :

- a2D Point or 3D point ,
- AI Unit Position
- Player Unit Position
- Static Unit Position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject
radius	number	Radius for sphere

Return values:

Without error :	boolean	True / false
If error	nil	

Object:isNear(reference, radius, deltaAltitude)

Description : test if a reference is close to the unit of the player. The zone is defined by a cylinder represented by a horizontal circle, the two sides of the cylinder correspond to an altitude difference taking the unit of the player for center.

Reference can be :

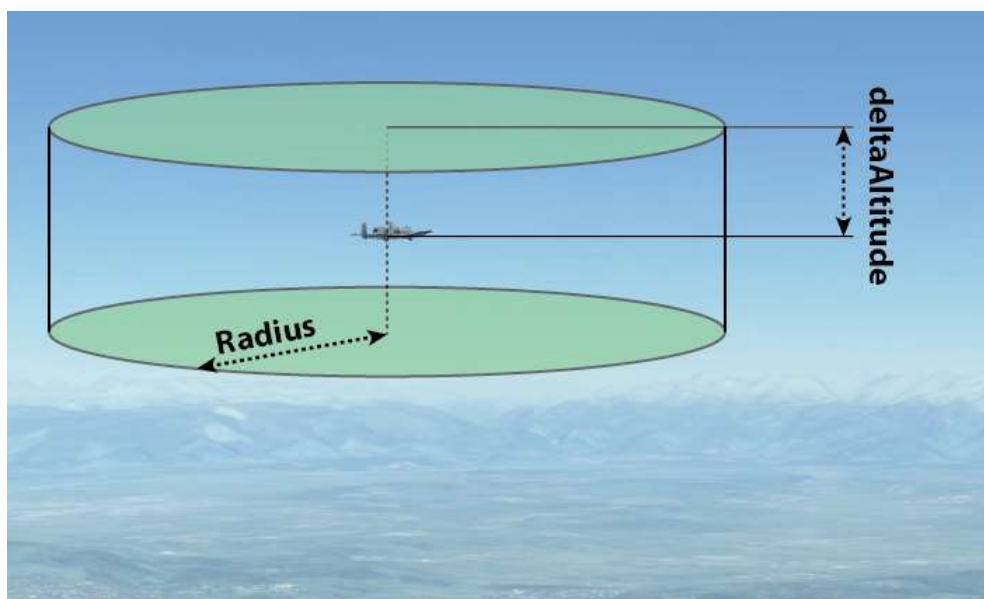
- a2D Point or 3D point ,
- AI Unit Position
- Player Unit Position
- Static Unit Position

Parameters : None

reference	table	3D Point x, y et z, or 2D Point x,y AI unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static Object Type ATME.C_StaticObject
radius	number	Radius
deltaAltitude	number	altitude distance in m

Return values:

Without error :	boolean	True or false
If error	nil	



Object:isPersonnelCarrier()

Description : This method test if the player's unit is transporting troops.

Parameters : None

Return values:

	boolean	True or false
--	---------	---------------

Object:isPlane()

Description : This method test if the player's unit is a plane.

Parameters : None

Return values:

Without error :	boolean	True or false
-----------------	---------	---------------

Object:load(groupToBoard, radius)

Description : This method makes it possible to embark a group of infantry. The player's unit must be a transport vehicle which have a capacity of transport of troops (personnel carrier). The radius specifies the radius available for troops to embark. The troop carrier unit must also be in zone with an horizontal speed less than 1m / s.

A troop transport vehicle has a maximum boarding capacity of infantry units. It can carry several infantry groups as long as the maximum capacity is not exceeded. The total mass of transport is currently unchanged.

The **ATME Core** event "**TRANSPORT_END_OF_BOARDING**" is adresssed at the end of embark process

Parameters :

groupToBoard	table	(ATME.C_Group) infantry group to embark
radius	number	Embark limit radius

Return values:

Without error :	boolean	False if no error
If error	boolean	True if errors
	string	Error codes :
		"LOAD_UNIT_TOO_FAR"
		"LOAD_UNIT_IN_AIR"
		"LOAD_UNIT_BAD_SPEED"
		"LOAD_TOO_MUCH_INFANTRY"
		"LOAD_INFANTRY_GROUP_NOT_READY"
		"LOAD_NO_INFANTRY_GROUP_AVAILABLE"

Allowed to transport troops helicopters :

"UH-1H" 8 infantry units limit, "Mi-8MT" 16 infantry units limit, "SA342" 2 infantry unit limit,

Allowed to transport troops ground vehicles :

"AAV7" 25 infantry units limit, "M-113" 11 infantry units limit, "LAV-25" 6 infantry units limit, "M1126 Stryker ICV" 9 infantry units limit, "M-2 Bradley" 6 infantry units limit, "BTR-80" 9 infantry units limit, "BTR_D" 12 infantry units limit, "MTLB" 10 infantry units limit, "BMD-1" 4 infantry units limit, "BMP-1" 8 infantry units limit, "BMP-2" 7 infantry units limit, "BMP-3" 7 infantry units limit, "UAZ-469" 6 infantry units limit, "Tigr_233036" 6 infantry units limit, "GAZ-3307" 16 infantry units limit, "GAZ-3308" 16 infantry units limit, "Ural-4320T" 20 infantry units limit, "Ural-4320-31" 20 infantry units limit, "M 818" 20 infantry units limit, "KAMAZ Truck" 20 infantry units limit

Object:soundOnce(file)

Description: plays a sound file for a specific player. The sound file must exist in the mission.

Parameters :

file	string	Sound files name
-------------	--------	------------------

Return values: Nonee

Object:soundChange(file, interval, forced)

Description : This method allows to modify the sound files already broadcast in a sound loop for a specific player. The change of list is done immediately (forced to true) or at the end of the reading of the file (forced = false). To stop playback, the soundStop function must be used. The sound file must exist in the mission.

Parameters :

file	string	Sound file name
interval	number	Interval in seconds
forced	boolean	True for immediate action. False for next file in the list

Return values: None

Object:soundPause(duration)

Description : This method allows to suspend active playback loop for a specified amount of time. At the end of the pause, playback automatically resumes beginning of file. The beginning of the file will be the following playlist file. This function is useful for occasionally inserting another sound into a loop playback.

Parameters :

duration	number	Duration of pause
-----------------	--------	-------------------

Return values: None

Object:soundStart(file, interval)

Description : This method allows to play a sound file at a regular intervals for a specific player.. To stop playback, the soundStop function must be used. The sound file must exist in the mission.

Parameters :

file	string	Sound file name
interval	number	Interval in seconds

Return values: None

Object:soundStop()

Description : stop active sound diffusion

Parameters : None

Return values: None

Object:soundChangePlayList(playList, interval, forced)

Description : This method allows you to modify a list of sound files already broadcast for a specific player. The playback mode (sequential or random) does not change. The interval sets the number of seconds of playback before moving to the next file. The change of list is done either immediately (forced to true) or at the end of the reading of the file (forced = false). To stop playback, the soundStop function must be used. Sound files must exist in the mission.

Parameters :

playList	table	Indexed list of sound file to play {[1] = xxx.ogg, ...}
interval	number	Interval in second (same for all)
forced	boolean	True for immediate modification false to wait next playlist

Return values: None

Object:soundStartPlayList(playList, randomRead, interval)

Description : This method allows you to play a list of sound files for a specific player. This reading can be sequential or random. In the case of sequential playback, after reading the last file in the list, playback resumes on the first file. The interval sets the number of seconds of playback before moving to the next file. The passage is forced by stopping the reading of the file course diffusion. To stop playback, the **soundStop** function must be used. Sound files must exist in the mission.

Parameters :

playList	table	Indexed list of sound file to play {[1] = xxx.ogg, ...}
randomRead	boolean	True for random play, false for sequential
interval	number	Interval in second (same for all files)

Return values: None

Object: unload(id)

Description : used to disembark an infantry group present in the player's transport Unit. Id corresponds to the index in the order of embark. it could be necessary to use **getGroupsNameOnBoard** in order to retrieve the indexed list of the troops on board. Id must match one of the indexes in that list of names.

ATME Core event "**TRANSPORT_END_OF_DISEMBARK**" is addressed at the end of the disembark

Parameters :

id	number	Index of boarding
-----------	--------	-------------------

Return values:

Without error :	boolean	False if no errors
	table	Unload Group instance table
If error	number	Lost Unit in Unload
	boolean	True if errors
	string	Errors code :
		"UNLOAD_UNIT_IN_AIR"
		"UNLOAD_UNIT_BAD_SPEED"
		"UNLOAD_BAD_ID"

Object:whichSide(reference)

Description : test the relative position (right or left) of the reference regards to the Player's unit. The orientation is fixed by the longitudinal axis and the speed vector (see diagram below with two example references).

A reference can be :

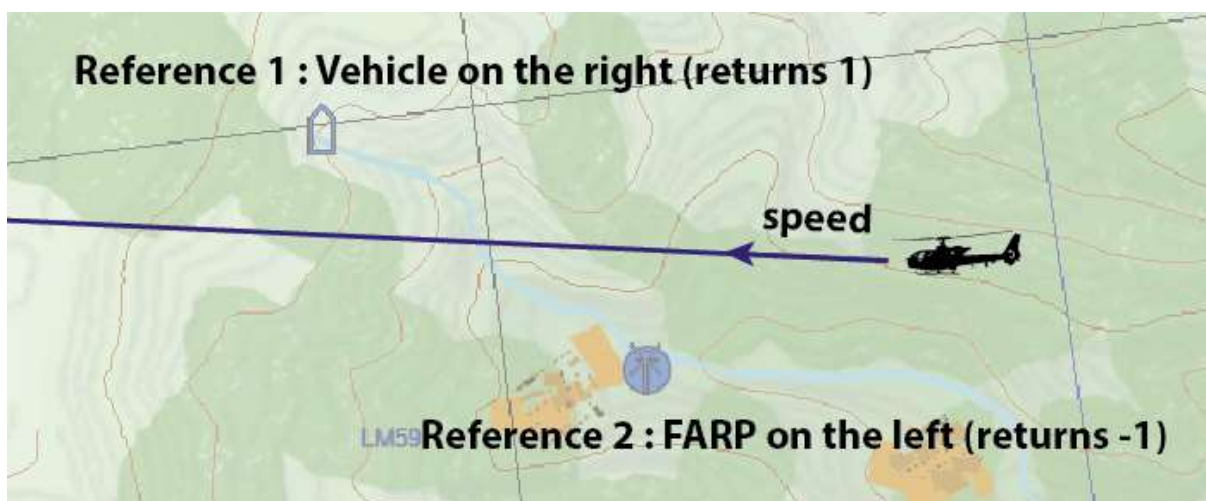
- a2D Point or 3D point
- AI Unit Position
- Player Unit Position
- Static Unit Position

Parameters :

reference	table	3D Point x, y et z, or 2D Point x,y AI Unit Type ATME.C_AIUnit Player Unit Type ATME.C_Player Static objec Type ATME.C_StaticObject
------------------	-------	---

Return values:

Without error :	number	-1 : reference is on the left side of the player 1 : reference is on the right side of the player 0 : reference is on longitudinal axe of the player
If error	nil	



Classe ATME.C_Race

This class allows to define and manage races between players. A race is defined by a set of doors with a left picket and a right picket. The left and right picket must be static objects with a unique name followed by "#" and an incremental number from 1 to 100. These names will be defined directly in the mission editor. The right and left picket number must be the same and set the door number. The distance between the two posts of the same door must be less than 500m.

The gate with the smallest number is the departure gate. The arrival door has the largest number. The doors are crossed in a precise order, from the door having the smallest number to the door having the largest number.

The number do not have to be consecutive. Also, a door will have an index number corresponding to the incremental number defined in the mission editor. The index number may be different and will correspond to the door number in the race:

- If gate # 003 is missing in the mission editor, gate "piquetG # 003" and "piquetD # 003", then the third gate of the race will be gate # 004.

in the mission editor there should be only one space between the name and the #.

race times are expressed in second with 2 decimal, it's a resolution of 1 / 100th of a second. A race generates Core events for each registered player.

ATME.C_Race(name, leftSideName, rightSideName, nbTurns)
--

Description : This function create a new race

Parameters :

name	string	Race name
leftSideName	string	General name without #XXX it is the static object representing the left picket.
leftRightName	string	General name without #XXX it is the static object representing the right picket.
NbTurns	number	Laps number for the race

Return values:

Without error	table	C_Race new instance
Si erreur	nil	

ATME.C_Race.exists(name)

Description : This Function test if a race exist.

Parameters :

name	string	Race name
-------------	--------	-----------

Return values:

Without error :	boolean	True if C_Race instance exist, else false
If error	nil	

ATME.C_Race.getByName(name)

Description : retrieve a race by its name

Parameters :

name	string	Race name
-------------	--------	-----------

Return values:

Without error :	table	C_Race instance or nil if doesn't exist.
If error	nil	

Object:addPlayer(player)

Description : This method subscribe a player to an existing race

Parameters :

player	table	C_Player instance (player)
---------------	-------	----------------------------

Return values: Nonee

Object:delete()

Description : This method removes the race. This results in the deletion in ATME Core of all this race elements. It will no longer be seen by ATME. The tables pointing to this race should be set to nil in the module (s) concerned because the destroyed instance will no longer be usable.

Parameters : None

Return values: None

Object:displayForAllPlayers(text, duration)

Description : This method displays a message for all players in a race. The message will be displayed for the specific duration.

Parameters :

text	string	Text to display
duration	number	Duration of the displayed message

Return values: Nonee

Object:isPlayerInRace(player)

Description : Test if a specific player is in a race

Parameters :

player	table	C_Player instance (player)
---------------	-------	-----------------------------

Return values:

	boolean	True / false
--	---------	--------------

Object:getName()

Description : This method retrieve the Object ATME class name, in this case "**C_Race**". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getDoorIndex(offset)

Description : This method returns the index of a door whose number is known. The door number is the number in the race. If the indexes are consecutive, the door number will be equivalent to the index. If the index numbers are not consecutive, the door number will be different. the door number in race is always consecutive from the first door (offset = 1) to the last door (offset corresponding to the number of doors defined).

Parameters :

offset	number	Gate number in race
---------------	--------	---------------------

Return values:

number	Gate index in mission editor
--------	------------------------------

Object:getDoorSides(index)

Description : This method returns the static left and right objects for a specific gate index. As a reminder, the index is the number fixed in the mission editor.

Parameters : None

Return values:

table	C_StaticObject instance left picket
table	C_StaticObject instance right picket

Object: getLapAtDoor(index)

Description : This method indicates if the gate, whose index is parameter, is an intermediate time check or not.

Parameters :

index	number	Gate index as defined in mission editor
--------------	--------	---

Return values:

boolean	True / false
---------	--------------

Object: getMaxAltitude()

Description : This method recover the maximum ground altitude (AGL) allowed when crossing a door gate.

Parameters : None

Return values:

number	Max allowed Altitude AGL
--------	--------------------------

Object: getMaxAltitudePenalty()

Description : This method allows to recover the penalty in seconds applied when exceeding the maximum ground altitude (AGL) allowed when crossing a door gate.

Parameters : None

Return values:

number	Penalty in seconds
--------	--------------------

Object: getMissedDoorPenalty()

Description : recover the penalty in seconds applied in case of of a door gate missed.

Parameters : None

Return values:

number	Penalty in seconds
--------	--------------------

Object: getName()

Description : retrieves race name

Parameters : None

Return values:

string	Race name
--------	-----------

Object: getNbDoors()

Description : retrieve doors gates quantity in a specific race

Parameters : None

Return values:

number	Gate doors quantity
--------	---------------------

Object: getNbTurns()

Description : retrieves number of laps from a specific race

Parameters : None

Return values:

number	Number of laps
--------	----------------

Object:getPlayerNextDoorIndex(player)

Description : this method gives the next door index to cross for a specific player

Parameters :

player	table	C_Player instance (player)
---------------	-------	-----------------------------

Return values:

	number	Next door index
--	--------	-----------------

Object:getRanking()

Description : returns the players actual ranking in race. It is an indexed table with fields for each entry:

- player : (instance C_Player)
- bestTime : best time in second with 2 digits.

Parameters : None

Return values:

	table	Indexed ranking table
--	-------	-----------------------

Object:removeAllPlayers()

Description : This method cancels the registration of all registered players in the race.

Parameters :

player	table	C_Player instance (player)
---------------	-------	----------------------------

Return values: Nonee

Object:removePlayer(player)

Description : this method cancel registration to a race for a specific player

Parameters :

player	table	C_Player Instance (player)
---------------	-------	----------------------------

Return values: None

Object:resetMaxAltitudeRule()

Description : This method removes the maximum allowed altitude control (at gate only) and the associated penalty.

Parameters : None

Return values: None

Object:resetMissedDoorRule()

Description : This method removes penalty rules applied if a player misses a door.

Parameters : None

Return values: None

Object:setLapAtDoor(index)

Description : This method defines the gate (index in parameter) where an intermediate time check is done.

Parameters :

index	number	Gate index as defined in mission editor
--------------	--------	---

Return values: None

Object:setMaxAltitudeRule(maxAltitude, penalty)

Description : This method defines the maximum allowed ground altitude (AGL) for crossing a door and the second penalty applied if a player exceeds that altitude.

This penalty does not apply if the player misses the door.

Parameters :

maxAltitude	number	Max AGL Altitude
penalty	number	Penalty in seconds

Return values: None

Object:setMissedDoorRule(penalty)

Description : This method defines penalty in seconds if a player miss a door gate

Parameters :

penalty	number	Penalty in seconds
----------------	--------	--------------------

Return values: None

Object : soundForAllPlayers(file)

Description : This method play a specific sound to all player in a Race

Parameters :

file	string	Sound file name
-------------	--------	-----------------

Return values: None

ATME.C_Smoke class

ATME.C_Smoke(colorName, point)

Description : This function creates smoke on a map point that will be activated later.

Parameters :

colorName	string	Smoke colour : "BLUE", "GREEN", "RED", "WHITE" et "ORANGE". "RANDOM"
point	table	3D Point x, y et z.

Return values:

Without error	table	C_Smoke new instance
Si erreur	nil	

Object:activate(restart)

Description : This method activates the smoke. If restart is true, the smoke will be automatically reactivated after 300s (5 minutes) and a stop will be necessary to stop it.

Parameters :

restart	boolean	If true, the smoke will continue after 5min. If false, stops after 5 min
----------------	---------	--

Return values: None

Object:getName()

Description : This method retrieve the object ATME class name, in this case "**C_Smoke**". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getColor()

Description : this method retrieve smoke colour.

Parameters : None

Return values:

string	Smoke colour : "BLUE", "GREEN", "RED", "WHITE" ou "ORANGE"
--------	--

Object:getPoint()

Description : retrieve the point where the smoke is defined

Parameters : None

Return values:

table	3D Point x, y et z, or 2D Point x,y
-------	-------------------------------------

Object:stop()

Description : This method removes subsequent reactivations. The smoke will stop after the current activation, (no more than 5 min).

Parameters : None

Return values: None

ATME.C_StaticObject class

The instances of this class are created and deleted by ATME Core.

ATME.C_StaticObject.exists(name)

Description : checking if a static object exists ("is alive") at a t time

Parameters :

name	string	Static object name
-------------	--------	--------------------

Return values:

Without error :	boolean	True if C_StaticObject instance exist else false
If error	nil	

ATME.C_StaticObject.getByName(name)

Description : Function to retrieve, from its name, a static "alive" object at a t time

Parameters :

name	string	Static object name
-------------	--------	--------------------

Return values:

Without error :	table	C_StaticObject instance
If error	nil	

Object:explode(power)

Description : triggers an explosion of a defined force on the static object.

Parameters :

power	number	Explosion power
--------------	--------	-----------------

Return values: Nonee

Object:fireFlare(color)

Description : triggers flare firing from the static object.

Parameters :

color	string	Flare colour "GREEN", "RED", "WHITE" et "YELLOW". "RANDOM"
--------------	--------	---

Return values: None

Object:fireIlluminationBomb(power)

Description : triggers a illumination bomb firing from the static object.

Parameters :

power	number	Bomb power
--------------	--------	------------

Return values: None

Object:fireSmoke(color)

Description : trigger a smoke signal near the static object

Parameters :

color	string	Smoke colour "BLUE" , "GREEN", "RED", "WHITE" et "ORANGE » "RANDOM"
--------------	--------	---

Return values: None

Object:getAzimuth()

Description : retrieve the azimuth (relative to the north) of the static object. This corresponds to the true heading without taking into account the magnetic deviation.

Parameters : None

Return values:

number	Azimuth in degrees (0-360)
--------	----------------------------

Object:getClassName()

Description : retrieve the object ATME class name, in this case " **C_StaticObject**". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getCoalitionName()

Description : Cette méthode permet de récupérer le nom de la coalition de l'unité AI.

Parameters : None

Return values:

Without error :	string	Nom de la coalition de l'unité : "RED", "BLUE" ou "NEUTRAL"
-----------------	--------	---

Object: getCountryName()

Description : this method retrieve the country from the static object

Parameters : None

Return values:

string	Country name
--------	--------------

Available country names (from DCS)

"RUSSIA", "UKRAINE", "USA", "TURKEY", "UK", "FRANCE", "GERMANY", "AGGRESSORS", "CANADA", "SPAIN", "THE_NETHERLANDS", "BELGIUM", "NORWAY", "DENMARK", "ISRAEL", "GEORGIA", "INSURGENTS", "ABKHAZIA", "SOUTH_OSETIA", "ITALY", "AUSTRALIA", "SWITZERLAND", "AUSTRIA", "BELARUS", "BULGARIA", "CHEZH_REPUBLIC", "CHINA", "CROATIA", "EGYPT", "FINLAND", "GREECE", "HUNGARY", "INDIA", "IRAN", "IRAQ", "JAPAN", "KAZAKHSTAN", "NORTH_KOREA", "PAKISTAN", "POLAND", "ROMANIA", "SAUDI_ARABIA", "SERBIA", "SLOVAKIA", "SOUTH_KOREA", "SWEDEN", "SYRIA"

Object: getDCSStaticObject()

Description : retrieves the static object as handled by DCS (Unit Class Object).

Parameters : None

Return values:

table	DCS class instance Unit
-------	--------------------------------

Object: getLife()

Description : method to retrieves the "life" information from an AI unit.

Parameters : None

Return values:

number	Life information
--------	------------------

Object: getName()

Description : retrieve the static object name.

Parameters : None

Return values:

string	static object name.
--------	---------------------

Object: getPosition()

Description : retrieves static objec position (Point 3D).

Parameters : None

Return values:

table	3D Point x, y et z
-------	--------------------

Object:getName()

Description : retrieves object type name from the static object.

Parameters : None

Return values:

string	Static object type name : "Ships", "Planes", "Helicopters", "Warehouses", "Cargos", "Unarmed", "Fortifications"
--------	---

ATME.C_UserTrigger class

This class instances are created from the following two functions of the ATME class.C_Module:

- **createAbsoluteUserTrigger**
- **createFlagRelativeUserTrigger**

A user trigger is associated with a single module which created it. It is also defined by its name which can exist in two different modules without interaction. At any given time, there can not be two triggers created with the same name in a given module.

These triggers operate on the basis of relative or absolute time. They will thus trigger over a defined period and will then be automatically destroyed, allowing to create a new trigger of the same name later.

Object:getName()

Description : retrieves the object ATME class name of the object, in this case "**C_UserTrigger**". This method exists for all classes.

Parameters : None

Return values:

string	ATME class name
--------	-----------------

Object:getFlag()

Description : This method retrieves the associated flag with a user trigger created with the **createFlagRelativeUserTrigger** function.

Parameters : None

Return values:

number	Linked Flag index, -1 if no linked flag
--------	---

Object:getName()

Description : retrieves the name of a scheduled and / or activated trigger.

Parameters : None

Return values:

string	Trigger name
--------	--------------

Object:getTimeEnd()

Description : retrieves the number of seconds corresponding to the end of activation of the user trigger.

If the user trigger was created with the **C_Module createFlagRelativeUserTrigger** function, this value will only be significant if the trigger has been armed (see **isArmed** function).

Parameters : None

Return values:

number	Number of second
--------	------------------

Object:getTimeStart()

Description : retrieves the number of seconds corresponding to the activation of the user trigger.

If the user trigger was created with the **C_Module createFlagRelativeUserTrigger** function, this value will only be significant if the trigger has been armed (see **isArmed** function).

Parameters : None

Return values:

number	Number of second
--------	------------------

Object:isActivated()

Description : test if a user trigger is active. It test if the mission time is between the trigger time start and the trigger time end.

Parameters : None

Return values:

boolean	True if armed trigger else false
---------	----------------------------------

Object:isArmed()

Description : method to determine if a user trigger defined with a flag has been armed.in fact if the associated flag has been activated.

Parameters : None

Return values:

boolean	True if trigger is armed
---------	--------------------------

ATME.C_Vector3D class

This class defines the management of three-dimensional vectors.

It is also possible to add or subtract two vectors.

ATME.C_Vector3D(...)

Description : This function creates a 3D vector from data Parameters in parameters.

Parameters cas 1 : no parameters : create a nul vector (0,0,0)

Parameters cas 2 : create a vector copy

vector	table	C_Vector3D Instance
---------------	-------	---------------------

Parameters cas 3 : Crée un vecteur à partir de deux points A et B (vecteur AB)

pA	table	Point A -3D Point x, y et z, or 2D Point x,y
pB	table	Point B -3D Point x, y et z, or 2D Point x,y

Parameters cas 4 : create a vector from Coordinates data x, y et z

x	number	Coordinates x
y	number	Coordinates y
z	number	Coordinates z

Return values:

Without error	table	C_Vector3D new instance
Si erreur	nil	

Object:get()

Description : recover the x, y and z coordinates from a 3D vector.

Parameters : None

Return values:

	number	Coordinates x
	number	Coordinates y
	number	Coordinates z

Object:getAzimuth()

Description : recover vector azimuth (with respect to the North).

Parameters : None

Return values:

Without error :	number	Azimuth in degrees (0-360)
If error	nil	

Object:getClassName()

Description : retrieves the object ATME class name, in this case "C_Vector3D". This method exists for all classes.

Parameters : None

Return values:

	string	ATME class name
--	--------	-----------------

Object: getHModule()

Description : recover the horizontal vector module, calculated as suit : $\sqrt{x^2 + z^2}$

Parameters : None

Return values:

Without error :	number	Horizontal module in m
If error	nil	

Object: getModule()

Description : recover the vector module, calculated as suit : $\sqrt{x^2 + y^2 + z^2}$

Parameters : None

Return values:

Without error :	number	Vector module m
If error	nil	

Object: getX()

Description : retrieves 3D x coordinates.

Parameters : None

Return values:

number	Coordinates x
--------	---------------

Object:getY()

Description : retrieves 3D y coordinates.

Parameters : None

Return values:

number	Coordinate y
--------	--------------

Object:getZ()

Description : retrieves 3D z coordinates.

Parameters : None

Return values:

number	coordinate z
--------	--------------

Object:toUnitVector()

Description : This method returns a unitary collinear vector to this 3D vector. The vector will be a zero vector if the current object is a null vector (module = 0).

Parameters : None

Return values:

table	C_Vector3D instance
-------	---------------------

Object:scalaireH(vector)

Description : This method returns the horizontal scalar product, computed on the x and z axes, between the current vector and the vector in parameters.

Parameters :

vector	table	C_Vector3D instance
---------------	-------	---------------------

Return values:

	number	horizontal scalar product
--	--------	---------------------------